# Direct3D9 初级教程

## 作者：祝晓鹰 余锋



文章目录：

**前言：**

这篇教程是为初学者准备的，只要会简单的 VC++编程以及一点立体几何的基础知识即可。通过它，可以了解到 Direct3D 的基本概念，学会绘制简单的几何图形，并掌握光源、材质和纹理的基本用法。

3D 绘图的实质就是在二维计算机屏幕上创建三维幻觉，为此要用到一些数学变换来建模和处理几何图形。我将尽可能用通俗易懂的语言来解释这些变换及其用法，避免涉及复杂的数学知识。

由于我也是初学 Direct3D 不久，难免有疏漏或错误之处，读者如果有什么意见或建议，请发信至 zhawk@sina.com 。

## 1 开发环境

本文选用 VC++ 6.0 做语言环境，建议安装 Service Pack5 补丁包。为了创建 Direct3D 程序，需要从微软网站下载安装 DirectX SDK，我用的是最新的 9.0 版，大概 200 多兆，网址 http://www.microsoft.com/downloads/ 。安装完 SDK 后，开发环境就搭建好了。当然，要运行编译好的程序，DirectX9 是必不可少的（微软已经在 SDK 中包含了 DirectX9 的安装文件）。

DirectX9 SDK 为 VC 用户提供了一个程序向导，可以很方便地生成“空”的 Direct3D 程序。不过为了便于读者掌握 Direct3D 编程的基础知识，本文将以 MFC 单文档程序（SDI）为框架，在其上添加 Direct3D 绘图功能。

## 2 使用 COM 组件

DirectX 的功能都是以 COM 组件的形式提供的。COM 是组件对象模型（Component Object Model）的简写，它是一种协议，用来实现软件模块间的二进制连接。当这种连接建立后，两个模块之间就可以通过称为“接口（Interface）”的机制来通信。我们常用的 ActiveX 控件就是一种 COM 组件。

COM 的实现细节相当复杂，完全可以写一本厚厚的专著。不过别担心，微软已经为我们最大限度地简化了 COM 的使用，即便你对 COM 一窍不通也没关系。作为 Direct3D 开发人员，只要了解接口及其用法就行了：所谓接口，其实就是一组特殊的 C++对象，应用程序通过调用这些对象的成员函数，来访问 COM 组件，实现组件的功能。在 COM 术语中，这些成员函数被称作方法（Method）。虽然称呼变了，但其调用语法与普通的 C++对象相比，并无二致。接口的特殊性在于它的生成和销毁都由系统完成，无须用户干预。

在 Direct3D 编程中，我们要做的工作基本上可以归纳为：

- 调用适当的函数获取接口指针；
- 调用接口的方法（成员函数）来完成所需功能；
- 用完接口后，调用 Release 方法进行“释放”，注意释放顺序应该和获取它们的顺序相反。

## 3 第一个程序－初始化 Direct3D

在本节中，我们将编写一个简单的 Direct3D 程序，它在 MFC 单文档程序的基础上，生成一个蓝色背景的 Direct3D 窗口。通过该例程，我们可以了解 Direct3D 的初始化过程。

### 3.1 创建程序框架

进入 VC，新建一个工程 d3d001，工程类型选“ MFC AppWizard (exe) ”，即 MFC 应用程序，然后把程序类型设置为“ Single document ”－单文档，其余选项使用缺省设置即可。之所以选择单文档程序，是为了借用 MFC 的界面和消息处理机制，至于文档和视图，这里用不上。

打开类向导，以 CWnd 为基类，派生一个窗口类 CD3DWnd。我们把它作为 Direct3D 窗口，这意味着绘制好的三维图形将显示在该窗口，而所有和 Direct3D 相关的代码也都放在该类中。

接下来添加一些代码，以便在程序运行时显示该窗口：

- 编辑工具条资源 IDR_MAINFRAME，增加两个按钮，命令 ID 分别设为 ID_D3D_BEGIN 和 ID_D3D_END；
- 为主窗口 CMainFrame 增加一个 CD3DWnd 类型的数据成员，代码如下（黑体为用户新输入的部分，下同）：

（MainFrm.h）

```
... ...
#include "D3DWnd.h"
class CMainFrame : public CFrameWnd
{
protected:
    CD3DWnd    m_wndD3D;
... ...
```

- 利用类向导，为 CMianFrame 添加工具按钮 ID_D3D_BEGIN 和 ID_D3D_END 的消息处理函数，前者用于创建并显示一个 CD3DWnd 窗口，后者则用来销毁它。代码如下：

（MainFrm.cpp）

```
void CMainFrame::OnD3dBegin()
{
    m_wndD3D.CreateEx(
        0, AfxRegisterWndClass(0,NULL,NULL,NULL),
        "Direct3D 窗口", WS_POPUP | WS_CAPTION | WS_VISIBLE,
        CRect(100,100,500,500), this, 0);
}

void CMainFrame::OnD3dEnd()
{
    m_wndD3D.DestroyWindow();
}
```

编译并运行程序，然后点击按钮 ID_D3D_BEGIN，会弹出一个窗口，点击另一个按钮，窗口消失。当然了，目前为止它还只是一个普通的 MFC 窗口，下面我们为它添加 Direct3D

功能。

请读者注意，本文中的程序均忽略了出错情况下的处理，这样做的目的是为了简化代码。在正式编程中，还是应该考虑进行适当的出错处理。

### 3.2 初始化 Direct3D

对于 Direct3D 程序来说，第一步要做的就是创建 Direct3D 对象，然后用该对象创建设备对象。设备对象可以说是 Direct3D 中最重要的部件，几乎所有的 3D 绘图功能都要通过它实现。

为类 CD3DWnd 添加下列成员：

（D3DWnd.h）

```
... ...
#include <d3d9.h>
#include <d3dx9math.h>
class CD3DWnd : public CWnd
{
protected:
    LPDIRECT3D9    m_pD3D;      //Direct3D 对象的接口指针
    LPDIRECT3DDEVICE9    m_pDevice;   //设备对象的接口指针
    void InitD3D();     //该函数用于初始化 Direct3D
... ...
```

输入初始化函数 InitD3D 的代码：

（D3DWnd.cpp）

```
void CD3DWnd::InitD3D()
{
    //创建 Direct3D 对象，并获取接口 IDirect3D9 的指针，
    //我们将通过该指针操作 Direct3D 对象。
    m_pD3D = ::Direct3DCreate9(D3D_SDK_VERSION);

    D3DPRESENT_PARAMETERS    d3dpp;
    ::ZeroMemory(&d3dpp, sizeof(d3dpp));
    d3dpp.Windowed = TRUE;       //创建窗口模式的 Direct3D 程序
    d3dpp.SwapEffect = D3DSWAPEFFECT_DISCARD;
    d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;

    //调用方法 IDirect3D9::CreateDevice 创建设备对象，并获取
    //接口 IDirect3DDevice9 的指针，我们将通过该指针操作设备对象
    m_pD3D->CreateDevice(
        D3DADAPTER_DEFAULT,      //使用缺省的显卡
        D3DDEVTYPE_HAL,          //指定设备类型为 HAL
        m_hWnd,                  //Direct3D 窗口的句柄
        D3DCREATE_SOFTWARE_VERTEXPROCESSING,//软件顶点处理
        &d3dpp, &m_pDevice);
}
```

作为 Direct3D 中的渲染部件，设备对象用于顶点变换、光照处理以及矢量图形的光栅化。Direct3D 目前支持两种设备类型：HAL（硬件抽象层 Hardware Abstraction Layer）和 Reference。前者启用硬件三维加速功能，需要显卡支持，如 NVIDIA 的 TNT、GeForce 系列；后者则以软件模拟方式完成三维处理，虽然速度慢，但可以在任意显卡上运行，一般用于调试程序，其对应的参数为 D3DDEVTYPE_REF。

上述代码使用了软件顶点处理方式，如果显卡支持硬件 T/L，如 GeForce 系列，可以选用硬件方式以提高效率，调用参数为 D3DCREATE_HARDWARE_VERTEXPROCESSING。

用类向导为 CD3DWnd 添加 WM_CREATE 的消息处理函数 OnCreate，在其中调用 InitD3D，以便在创建窗口的同时完成初始化工作：

（D3DWnd.cpp）

```
int CD3DWnd::OnCreate(LPCREATESTRUCT lpCreateStruct)
{
    if (CWnd::OnCreate(lpCreateStruct) == -1)
        return -1;
    InitD3D();      //初始化 Direct3D
    return 0;
}
```

### 3.3 渲染

初始化完成后，就可以开始执行渲染操作，即前面所说的绘图。

在 Direct3D 中，一个设备对象至少包含两个显示缓存区：当前缓存区（Front Buffer）和后备缓存区（Back Buffer），前者可以看成 Direct3D 窗口的映射。当我们渲染图形时，实际上并不是直接在窗口上输出，而是在后备缓存区上绘图。渲染完毕后，交换两个缓存区，使原来的后备缓存区变成当前缓存区，从而实现窗口刷新，如图 1 所示。快速重复此过程，就会在屏幕上形成连续的动画。

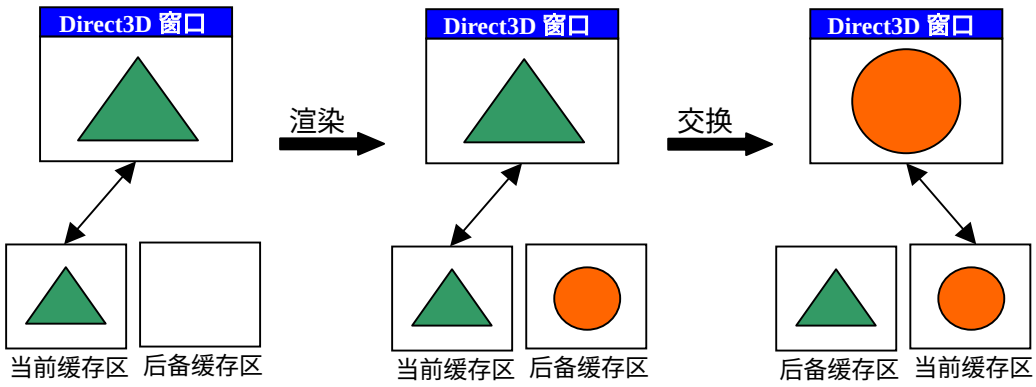


图 1

现在来编写渲染部分的代码，给 CD3DWnd 添加一个成员函数 Render：

（D3DWnd.h）

```
... ...
void InitD3D();
void Render();      //该函数用于渲染
... ...
```

（D3DWnd.cpp）

```
void CD3DWnd::Render()
{
    //用指定颜色清除后备缓存区
    m_pDevice->Clear(
        0, NULL, D3DCLEAR_TARGET,
        D3DCOLOR_XRGB(0,0,255),        //指定使用蓝色
        1.0f, 0);

    //Direct3D 规定在渲染前必须调用方法 IDirect3DDevice9::BeginScene，
    //结束时要调用 IDirect3DDevice9::EndScene。
    m_pDevice->BeginScene();

    //实际的渲染代码放在此处。因为本节只是为了演示如何初始化 Direct3D，
    //所以这里为空，生成的 Direct3D 窗口将是一个蓝色背景的空白窗口

    m_pDevice->EndScene();

    //交换当前/后备缓存区，刷新窗口
    m_pDevice->Present(NULL, NULL, NULL, NULL);
}
```

打开类向导，为 CD3DWnd 添加 WM_PAINT 的消息处理函数 OnPaint，在其中调用 Render：

（D3DWnd.cpp）

```
void CD3DWnd::OnPaint()
{
    CPaintDC dc(this);
    Render();      //渲染
}
```

### 3.4 释放接口

为 CD3DWnd 添加成员函数 Cleanup，输入“释放”代码：

（D3DWnd.h）

```
... ...
void Render();
void Cleanup();    //该函数用于释放接口
... ...
```

（D3DWnd.cpp）

```
void CD3DWnd::Cleanup()
{
    m_pDevice->Release();       //释放设备对象
```

```
        m_pD3D->Release();          //释放 Direct3D 对象
    }
```

用类向导为 CD3DWnd 添加 WM_DESTROY 的消息处理函数 OnDestroy，在其中调用 Cleanup。这样当窗口关闭时，就会自动释放接口：

（D3DWnd.cpp）

```
void CD3DWnd::OnDestroy()
{
    CWnd::OnDestroy();
    Cleanup();    //释放接口
}
```

现在代码已经全部输入完毕，不过在编译前，还需要为连接器指定 Direct3D 的导入库：选择 VC 的菜单项“ Project/Settings... ”，然后选中“ Link ”标签，在“ Object/library modules ”栏输入“ d3d9.lib d3dx9.lib ”（本节中的例程只用到了 d3d9.lib，另外一个库文件是为后面程序准备的）。

编译运行程序，点击按钮 ID_D3D_BEGIN，会弹出一个蓝色的 Direct3D 窗口（见图 2），点击另一个按钮，窗口消失。至此，我们的第一个 Direct3D 程序就算是大功告成了。


图 2

## 4 画一个三角形

在 Direct3D 中，所有三维实体都是由三角形构成的，本节将演示如何画一个三角形。事实上，从编程角度来看，画一个三角形和画一个由成百上千个三角形构成的复杂图形并没有什么区别。

### 4.1 一些数学概念

在开始编程之前，读者有必要了解一些有关三维坐标系的基本概念。

#### 4.1.1 三维坐标系、点、矢量

按坐标轴之间的相互关系划分，三维坐标系可分为左手坐标系和右手坐标系，如图 3 所示。在左手坐标系中，坐标轴的定义符合左手法则：左手四个手指的旋转方向从 X 轴到 Y 轴，大拇指的指向就是 Z 轴。右手坐标系依次类推。Direct3D 使用左手坐标系，其中 X 轴表示左右，Y 轴表示上下，Z 轴表示远近（深度）。

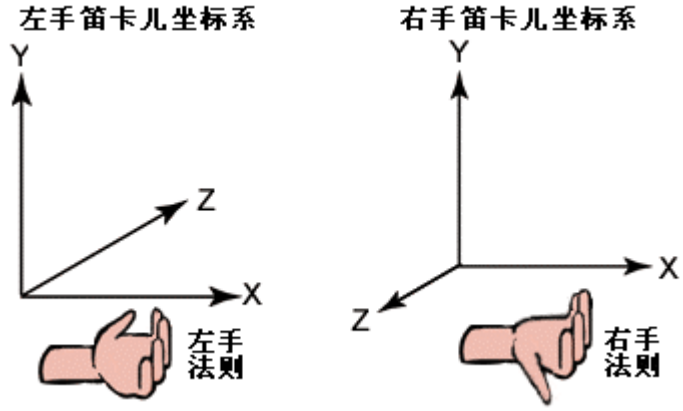


图 3

取定坐标系后，空间中的任意一点可以用一组坐标值（x，y，z）来表示。矢量是空间中的一条有向线段，Direct3D 用它来标识空间方向。矢量的表示方法与点坐标类似，也是用｛x，y，z｝，不过它表示的是从原点指向点（x，y，z）的有向线段。矢量和起点无关，只要两个矢量同向（平行）且等长，就认为它们相等。在 Direct3D 中，点和矢量通常使用同一个结构 D3DXVECTOR3 保存。

矢量的计算公式很简单：假设矢量的起点为 M（$x_1$，$y_1$，$z_1$），终点为 N（$x_2$，$y_2$，$z_2$），则矢量 $\overrightarrow{MN}$ = ｛$x_2$-$x_1$，$y_2$-$y_1$，$z_2$-$z_1$｝。

矢量除了方向属性外，也有大小（长度），但是 Direct3D 一般不用。为了避免矢量的大小给计算带来误差，可以用函数 D3DXVec3Normalize 把它变换成单位矢量（长度为 1）。

#### 4.1.2 三角形、平面法线、顶点法线

在 Direct3D 中，三角形是构成实体的基本单位，因为一个三角形正好是一个平面，以三角形面为单位进行渲染效率最高。大量的三角形组合在一起，构成复杂的多边形或者曲面。图 4 是一个球面的例子。

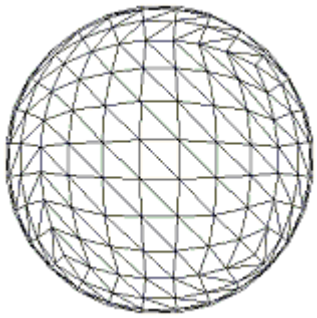

图 4

一个三角形由三个点构成，习惯上把这些点称为顶点（Vertex）。三角形平面有正、反面之分，由顶点的排列顺序决定：顶点按顺时针排列的表面是正面，如图 5 所示。其中与三角形平面垂直、且指向正面的矢量称为该平面的法线（Normal）。在 Direct3D 中，为了提高渲染效率，缺省条件下只有正面可见，不过可以通过 IDirect3DDevice9::SetRenderState 来改变设置，其对应的渲染状态常数为 D3DRS_CULLMODE，具体用法请参阅 SDK 文档。

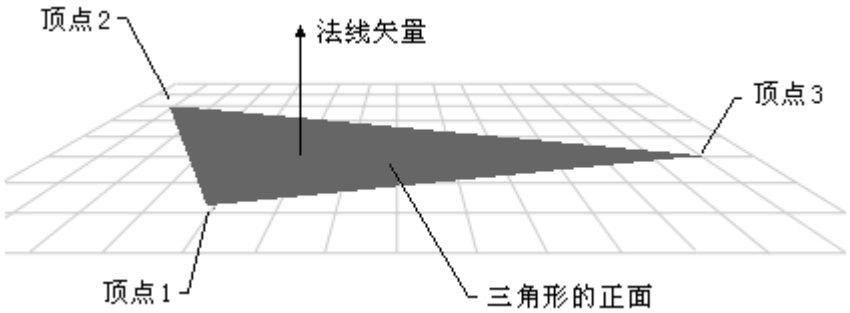


图 5

顶点法线（Vertex Normal）是过顶点的一个矢量，用于在高洛德着色（Gouraud Shading）中计算光照和纹理效果。在生成曲面时，通常令顶点法线和相邻平面的法线保持等角，如图 6-1 所示，这样进行渲染时，会在平面接缝处产生一种平滑过渡的效果。如果是多边形，则令顶点法线等于该点所属平面（三角形）的法线，如图 6-2 所示，以便在接缝处产生突出的边缘。

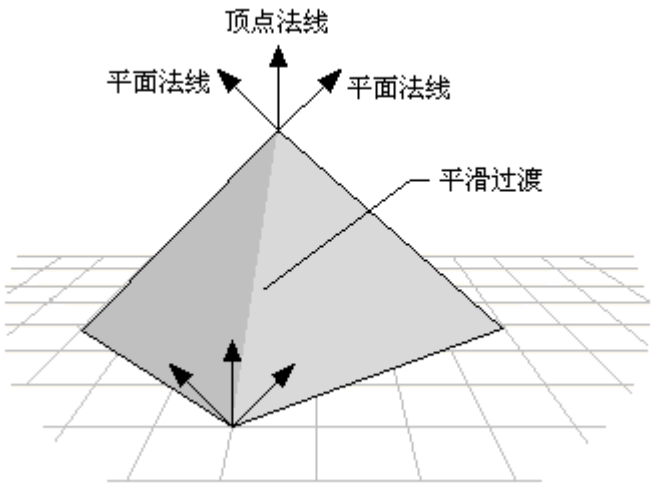


图 6-1

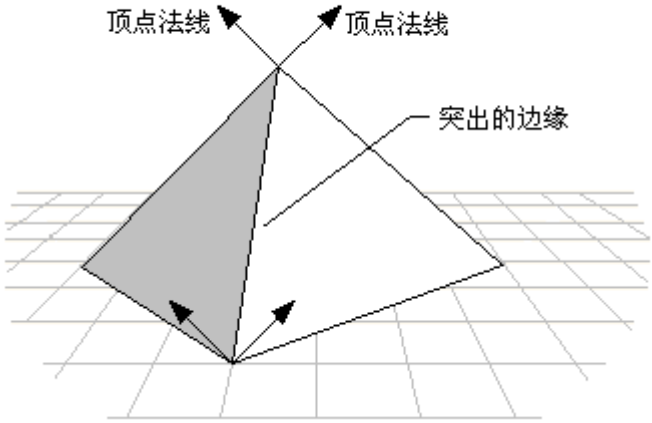


图 6-2

**4.1.3 Direct3D 设备支持的图元（Primitive）格式**

在 Direct3D 中，三维实体都是由一些基本图元组合而成的，总共有 6 种图元格式（示例参见图 7-1－6）：

- 点列（Point Lists）
  由顶点组成的集合；
- 线列（Line Lists）
  由直线段组成的集合；
- 线带（Line Strips）
  由互相连接的直线段组成的集合；
- 三角形列（Triangle Lists）
  由三角形组成的集合，每三个顶点构成一个三角形；
- 三角形带（Triangle Strips）
  由相接的三角形组成的集合。在例图中，v1、v2、v3 构成第一个三角形，v2、v3、v4 构成第二个三角形... ...（注意：三角形带的正面由第一个三角形决定，因此第二个三角形顶点的排列顺序实际上应该为 v2、v4、v3）；
- 三角扇形（Triangle Fans）
  由相接且共点的三角形组成，v1、v2、v3 构成第一个三角形，v1、v3、v4 构成第二个三角形... ...；


图 7-1

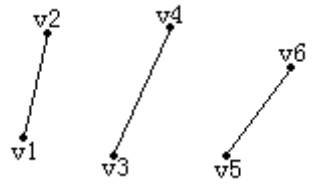


图 7-2

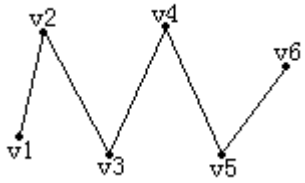


图 7-3

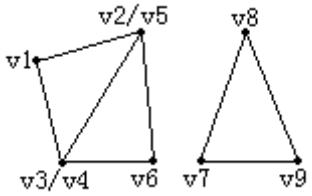


图 7-4

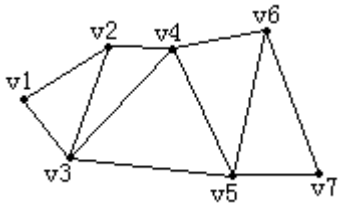


图 7-5

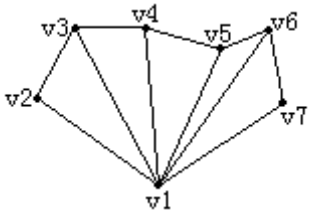


图 7-6

上述图元中，后三种以三角形为单位的图元比较常用。其中三角形列适用范围较广，既能用于多边形，也可用于曲面；而三角形带和三角扇形由于存在公共顶点，如果用来创建多边形，其公共顶点的法线不好确定，因此通常只用于曲面，不过在三角形数目相同的情况下，它俩使用的顶点数目要比前者少得多。

### 4.1.4 坐标变换

**1）世界变换**

我们在建立三维实体的数学模型时，通常以实体的某一点为坐标原点，比如一个球体，很自然就用球心做原点，这样构成的坐标系称为本地坐标系（Local Coordinates）。实体总是位于某个场景（World Space）中，而场景采用世界坐标系（World Coordinates），如图 8 所示，因此需要把实体的本地坐标变换成世界坐标，这个变换被称为世界变换（World Transformation）。

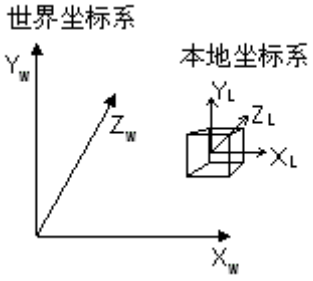


图 8

在 Direct3D 中，坐标变换通过一个 4x4 矩阵来实现，对于世界变换，只要给出实体在场景中的位置信息，就可以借助 Direct3D 函数得到变换矩阵，具体的计算步骤如下：

- 首先把实体放置在在世界坐标系原点，使两个坐标系重合；
- 在世界空间中，对实体进行平行移动，其对应的平移变换阵 $T_T$ 可由函数 D3DXMatrixTranslation 求得；
- 把平移后的实体沿自身的 Z 轴旋转一个角度（角度大于 0，表示从 Z 轴的正向朝原

点看去，旋转方向为顺时针；反之为逆时针，下同），对应的旋转变换阵 $T_Z$ 用 D3DXMatrixRotationZ 计算；

- 把实体沿自身的 Y 轴旋转一个角度，用 D3DXMatrixRotationY 求出变换阵 $T_Y$；
- 把实体沿自身的 X 轴旋转一个角度，用 D3DXMatrixRotationX 求出变换阵 $T_X$；
- 最后对实体进行缩放，假设三个轴的缩放系数分别为 $s_x$、$s_y$、$s_z$，该操作对应的变换阵 $T_S$ 可由函数 D3DXMatrixScaling 求得；
- 最终的世界变换矩阵 $T_W = T_S \cdot T_X \cdot T_Y \cdot T_Z \cdot T_T$ ，在 Direct3D 中，矩阵乘法用函数 D3DXMatrixMultiply 实现，注意相乘顺序为操作的逆序。

从以上描述中，我们很容易得出：实体的运动可以通过不断改变世界变换矩阵来实现。

**2）视角变换**

实体确定后，接下来要确定观察者在世界坐标系中的方位，换句话说，就是在世界坐标系中如何放置摄像机。观察者（摄像机）所看到的景象，就是 Direct3D 窗口显示的内容。

确定观察者需要三个量：

- 观察者的点坐标；
- 视线方向，为一个矢量，不过 Direct3D 用视线上的一个点来替代，此时视线方向就是从观察者指向该目标点，这样表示更直观一些；
- 上方向，通俗地说，就是观察者的头顶方向，用一个矢量表示。

确定后，以观察者为原点，视线为 Z 轴，上方向或它的一个分量为 Y 轴（X 轴可由左手法则得出，为右方向），构成了视角坐标系，如图 9 所示。我们需要把实体从世界空间转换到视角空间，这个坐标变换被称为视角变换（View Transformation）。

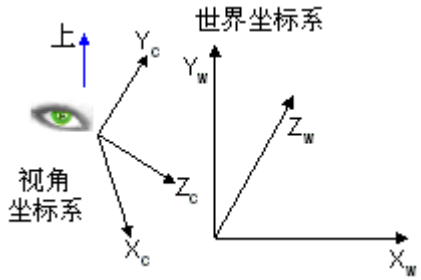


图 9

与世界变换相比，视角变换矩阵的获取要容易得多，只需调用一个函数 D3DXMatrixLookAtLH，其输入参数就是决定观察者的那三个量。

**3）投影变换**

实体转换到视角空间后，还要经过投影变换（Projection Transformation），三维的实体才能显示在二维的计算机屏幕上。打个比方，如果把屏幕看做照相机中的胶卷，那么投影变换就相当于照相机的镜头。

Direct3D 使用透视投影变换（Perspective Transformation），此时在视角空间中，可视区域是一个以视线为轴心的棱台（Viewing Frustum），如图 10 所示。想象一下你处在一个伸手不见五指的房间里，面前有一扇窗户，你可以透过窗户看到各种景物。窗户就是棱台的前裁剪平面，天空、远山等背景是后裁剪平面，其间的可视范围是景深。投影变换把位于可视棱台内的景物投影到前裁剪平面，由于采用透视投影，距离观察者远的对象会变小，从而更具有真实感。在 Direct3D 中，前裁剪平面被映射到程序窗口，最终形成了我们在屏幕上看到的画面。

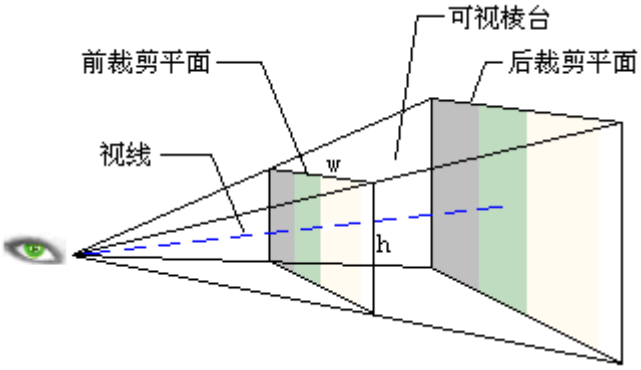


图 10

透视投影变换由四个量决定：

- 前裁剪平面的宽度 w；
- 前裁剪平面的高度 h；
- 前裁剪平面到原点的距离 z1；
- 后裁剪平面到原点的距离 z2。

由于 w、h 用起来不是很直观，因此实际应用中，常用 fov 和 aspect 代替 w、h，其中 fov 是 Y 方向上的可视角度，通常取π/4；aspect 是前裁剪平面的高度与宽度之比，通常取 1（由三角函数定义，易知 h=2 · z1 · tg(fov/2)，w=h/aspect）。用这四个量来调用函数 D3DXMatrixPerspectiveFovLH，即可获得投影变换矩阵。

得到三个变换矩阵后，还需要调用方法 IDirect3DDevice9::SetTransform 把它们设置到渲染环境中，具体用法参见后面的例程。

最后，可以用三句话来概括这些变换的作用：世界变换决定实体的位置；视角变换决定观察者的位置；投影变换决定观察者的可视区域。

至此，相关的数学部分终于讲完了，可能枯燥了点，但却是掌握 Direct3D 的关键。作为程序员，虽然不需要我们了解这些算法的来历、推导等，但一定要知道它们是干什么用的以及如何用。

### 4.2 画一个三角形

前面讲了一大堆抽象的理论，现在让我们理论联系实际，来画一个最简单的三角形。

#### 4.2.1 建模

该三角形在本地坐标系的数学模型如图 11 所示，三个顶点的本地坐标分别为 A（-1，-1，0）、B（0，1，0）、C（1，-1，0）。选取三角形列作为实体的图元格式，顶点排列顺序为 A、B、C，正面（可视面）朝外。

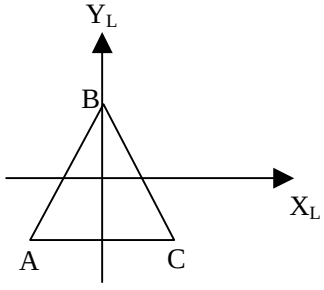


图 11

### 4.2.2 创建顶点缓存区

首先要定义顶点格式，Direct3D 采用了一种被称之为“可变形顶点格式 Flexible Vertex Format（(FVF)”的技术，除顶点坐标外，还可以包括顶点的法线、颜色、纹理等数据。在本节中，用到了坐标和颜色。通常情况下，实体的外观由材质、光照和纹理决定，不需要再另外为顶点定义颜色，但目前还没有讲到这些内容，因此要给出顶点的颜色。Direct3D 在渲染时，将使用顶点颜色，通过插值算法来填充三角形。

打开上一节生成的例程 d3d001，在 D3DWnd.cpp 的开始部分中加入 FVF 的定义：

（D3DWnd.cpp）

```
... ...
#include "D3DWnd.h"
//定义 FVF 的顶点结构
struct CUSTOMVERTEX
{
    float x, y, z;          //顶点坐标
    DWORD color;    //顶点颜色
};
//定义 FVF 用到的数据项：坐标 颜色
#define D3DFVF_CUSTOMVERTEX (D3DFVF_XYZ | D3DFVF_DIFFUSE)
... ...
```

在 Direct3D 中，FVF 顶点数据按图元的格式，顺序存放在顶点缓存区（Vertex Buffer），它是一个 COM 对象，通过接口 IDirect3DVertexBuffer9 访问。以下是创建顶点缓存区的代码：

（D3DWnd.h）

```
... ...
void Cleanup();

LPDIRECT3DVERTEXBUFFER9 m_pVB;    //顶点缓存区的的接口指针
void InitGeometry();    //该函数用于建模
... ...
```

（D3DWnd.cpp）

```
void CD3DWnd::InitGeometry()
{
    //三角形实体的数学模型
    CUSTOMVERTEX vertices[] =
        {{ -1.0f, -1.0f, 0.0f, D3DCOLOR_XRGB(255,0,0) },  //点 A，红色
        {    0.0f, 1.0f, 0.0f, D3DCOLOR_XRGB(0,255,0) },   //点 B，绿色
        {    1.0f, -1.0f, 0.0f, D3DCOLOR_XRGB(0,255,255) }};//点 C，浅蓝

    //创建顶点缓存区，并获取接口 IDirect3DVertexBuffer9 的指针
    m_pDevice->CreateVertexBuffer(
        sizeof(vertices),     //缓存区尺寸
        0, D3DFVF_CUSTOMVERTEX,
```

```
                D3DPOOL_DEFAULT, &m_pVB, NULL );

        //把顶点数据填入顶点缓存区
        void* pVertices;
        m_pVB->Lock( 0, sizeof(vertices), (void**)&pVertices, 0 );
        memcpy( pVertices, vertices, sizeof(vertices) );
        m_pVB->Unlock();
    }
```

修改 CD3DWnd::OnCreate，加入对 InitGeometry 的调用：

（D3DWnd.cpp）

```
... ...
InitD3D();
InitGeometry();    //进行建模
... ...
```

由于顶点缓存区是 COM 对象，还要在 CD3DWnd::Cleanup 中添加它的释放代码：

（D3DWnd.cpp）

```
... ...
m_pVB->Release();     //释放顶点缓存区
m_pDevice->Release();
... ...
```

#### 4.2.3 设置变换矩阵

在本例中，我们把实体放到世界坐标系的原点，让它绕 Y 轴做顺时针旋转，为此要用到一个定时器，在 WM_TIMER 的处理函数中不断改变旋转角度。

对于视角变换，观察点定在（0，3，-5），视线目标点取原点，上方向取 Y 轴的正向，对应矢量为｛0，1，0｝。

投影变换的可视角取π/4，高宽比取 1，两个裁剪平面的距离分别取 1 和 100。

下面是生成变换矩阵的代码：

（D3DWnd.h）

```
... ...
void InitGeometry();

int m_nRotateY;   //实体的旋转角度（单位：度）
void SetupMatrices();  //该函数用于设置三个变换矩阵
... ...
```

（D3DWnd.cpp）

```
void CD3DWnd::SetupMatrices()
{
    float angle = m_nRotateY * D3DX_PI / 180; //把旋转角换算成弧度
    D3DXMATRIX matWorld;
    //计算世界变换矩阵
```

```
        ::D3DXMatrixRotationY( &matWorld, angle );
        //把世界变换矩阵设置到渲染环境
        m_pDevice->SetTransform( D3DTS_WORLD, &matWorld );

        D3DXVECTOR3 eye( 0.0f, 3.0f,-5.0f );      //观察点
        D3DXVECTOR3 lookat( 0.0f, 0.0f, 0.0f ); //视线目标点
        D3DXVECTOR3 up( 0.0f, 1.0f, 0.0f );       //上方向
        D3DXMATRIX matView;
        //计算视角变换矩阵
        ::D3DXMatrixLookAtLH( &matView, &eye, &lookat, &up );
        //把视角变换矩阵设置到渲染环境
        m_pDevice->SetTransform( D3DTS_VIEW, &matView );

        D3DXMATRIXA16 matProj;
        //计算透视投影变换矩阵
        ::D3DXMatrixPerspectiveFovLH( &matProj, D3DX_PI/4, 1.0f, 1.0f, 100.0f );
        //把投影变换矩阵设置到渲染环境
        m_pDevice->SetTransform( D3DTS_PROJECTION, &matProj );
    }
```

在 CD3DWnd::OnCreate 中添加定时器的初始化语句：

（D3DWnd.cpp）

```
... ...
InitD3D();
InitGeometry();
m_nRotateY = 0;
SetTimer( 1, 40 ,NULL );          //定时间隔设为 40 毫秒
... ...
```

用类向导为 CD3DWnd 添加 WM_TIMER 的消息处理函数 OnTimer，在其中累加旋转角度：

（D3DWnd.cpp）

```
void CD3DWnd::OnTimer(UINT nIDEvent)
{
    m_nRotateY += 2;       //每次旋转 2 度
    CWnd::OnTimer(nIDEvent);
}
```

#### 4.2.4 渲染

修改 CD3DWnd::Render，在其中加入三角形的绘制语句：

（D3DWnd.cpp）

```
... ...
m_pDevice->BeginScene();
SetupMatrices();  //设置变换矩阵
```

```
//设置自定义的 FVF
m_pDevice->SetFVF( D3DFVF_CUSTOMVERTEX );
//绑定顶点缓存区至设备数据源
m_pDevice->SetStreamSource( 0, m_pVB, 0, sizeof(CUSTOMVERTEX) );
//绘制图元，其中参数 1 为图元格式，参数 3 为三角形数目
m_pDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
m_pDevice->EndScene();
... ...
```

因为要让三角形旋转，所以 Render 的调用改放在时钟消息处理函数 OnTimer 中，原来的 OnPaint 函数用类向导予以删除：

（D3DWnd.cpp）

```
... ...
Render();      //渲染
m_nRotateY += 2;
... ...
```

最后，要禁用光照处理（缺省是打开的）。因为本例程使用顶点颜色进行渲染，如果不禁用的话，将会看到一个黑糊糊的三角形。另外，缺省情况下 Direct3D 只挑选三角形的正面进行渲染，当旋转到一定角度，背面朝向观察者时，图像会消失，因此要关闭该项特性。相关代码加在函数 CD3DWnd::InitD3D 中：

（D3DWnd.cpp）

```
... ...
m_pD3D->CreateDevice( ... ...
//因为使用顶点颜色渲染，所以要禁用光照处理
m_pDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
//关闭“挑选”功能，允许渲染背面
m_pDevice->SetRenderState( D3DRS_CULLMODE, D3DCULL_NONE );
... ...
```

编译运行程序，点击工具按钮 ID_D3D_BEGIN，将会出现一个旋转的彩色三角形，如图 12 所示。请读者妥善备份本节生成的例程，因为不仅是下一节，第 6 节也要用到它。

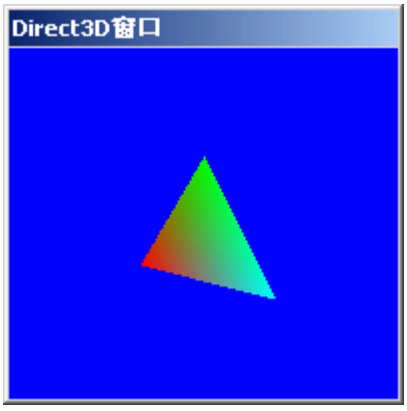


图 12

## 5 画一个三棱锥－索引缓存和 Z 缓存

本节将通过画一个三棱锥，介绍索引缓存（Index Buffer）和 Z 缓存（Z-Buffer）的用法。

### 5.1 什么是索引缓存

在 Direct3D 中，实体模型中的一个点可能被多个三角形面所共用，如图 13 所示的三棱锥，虽然只有 4 个顶点，却由 4 个三角形面组成。

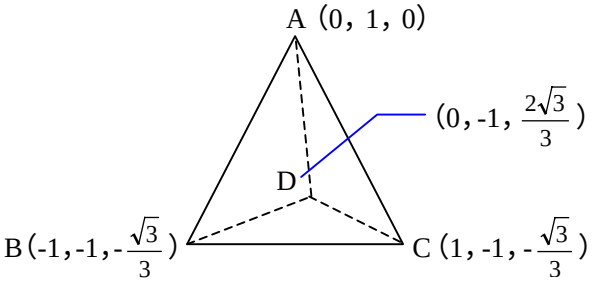


图 13

如果象上一节那样，把顶点数据按对应图元的格式，直接放进顶点缓存区，该棱锥使用三角形列，4 个锥面共需要 4 x 3 = 12 个顶点，也就是说，有 8 个顶点是重复的。如果实体比较复杂，重复的顶点会更多，造成资源浪费。

为此 Direct3D 引入了索引缓存的概念，把顶点的具体数据和代表图元格式的顶点顺序分开存储：顶点数据仍然放到顶点缓存区中，索引缓存区则按照图元格式，顺序存放顶点的索引。

以上面的棱锥的为例：首先在顶点缓存中保存 A、B、C、D 这 4 个顶点的 FVF 数据项，相应的索引为 0、1、2、3；然后按照三角形列的组成顺序，把顶点索引值存入索引缓存区，4 个三角形分别为△ACB、△ADC、△ABD、△BCD（注意顶点排列顺序和可视面的关系），则索引序列为 0 2 1　0 3 2　0 1 3　1 2 3。这样原本要用 12 个顶点数据构建一个三棱锥，使用索引缓存后，只需要 4 个。当然了，索引缓存本身也要占用一些资源，不过和节约的顶点缓存相比少多了。

### 5.2 创建索引缓存

打开上一节的例程，为 CD3DWnd 添加一个数据成员，用来保存索引缓存区的接口指针：

（D3DWnd.h）

```
... ...
void SetupMatrices();

LPDIRECT3DINDEXBUFFER9 m_pIB;        //索引缓存区的接口指针
... ...
```

修改函数 CD3DWnd::InitGeometry 中的建模部分，并添加索引缓存区的创建代码：

（D3DWnd.cpp）

```
void CD3DWnd::InitGeometry()
{
    //三棱锥的数学模型
    CUSTOMVERTEX vertices[] =        //FVF 顶点数据
```

```
        {{ 0.0f, 1.0f, 0.0f, D3DCOLOR_XRGB(0,255,0) },   //点 A，绿色
        { -1.0f, -1.0f, -0.577f, D3DCOLOR_XRGB(255,0,0) },   //点 B，红色
        { 1.0f, -1.0f, -0.577f, D3DCOLOR_XRGB(0,255,255) },  //点 C，浅蓝
        { 0.0f, -1.0f, 1.155f, D3DCOLOR_XRGB(255,0,255) }};  //点 D，粉红
    WORD    indices[] = { 0, 2, 1, 0, 3, 2, 0, 1, 3, 1, 2, 3 };      //索引序列

    //创建顶点缓存区，并获取接口 IDirect3DVertexBuffer9 的指针
    m_pDevice->CreateVertexBuffer(
        sizeof(vertices),     //缓存区尺寸
        0, D3DFVF_CUSTOMVERTEX,
        D3DPOOL_DEFAULT, &m_pVB, NULL );

    //把顶点数据填入顶点缓存区
    void* pVertices;
    m_pVB->Lock( 0, sizeof(vertices), (void**)&pVertices, 0 );
    memcpy( pVertices, vertices, sizeof(vertices) );
    m_pVB->Unlock();

    //创建索引缓存区，并获取接口 LPDIRECT3DINDEXBUFFER9 的指针
    m_pDevice->CreateIndexBuffer(
        sizeof(indices),      //缓存区尺寸
        0, D3DFMT_INDEX16,      //使用 16 bit 的索引值
        D3DPOOL_DEFAULT, &m_pIB, NULL );

    //把索引值填入索引缓存区
    void *pIndices;
    m_pIB->Lock( 0, sizeof(indices), (void**)&pIndices, 0 );
    memcpy( pIndices, indices, sizeof(indices) );
    m_pIB->Unlock();
}
```

还要在 CD3DWnd::Cleanup 中添加索引缓存区的释放代码：

（D3DWnd.cpp）

```
... ...
m_pIB->Release();      //释放索引缓存区
m_pVB->Release();
... ...
```

### 5.3 渲染索引缓存

由于使用了索引缓存，因此函数 CD3DWnd::Render 中的渲染部分也要进行相应修改：

（D3DWnd.cpp）

```
... ...
m_pDevice->SetStreamSource( 0, m_pVB, 0, sizeof(CUSTOMVERTEX) );
//绑定索引缓存区
```

```
m_pDevice->SetIndices( m_pIB );
//从索引缓存区绘制图元，参数 1 为图元格式，参数 4 为顶点数，参数 6 为三角形数
m_pDevice->DrawIndexedPrimitive( D3DPT_TRIANGLELIST, 0, 0, 4, 0, 4 );
m_pDevice->EndScene();
... ...
```

编译运行程序，然后弹出 Direct3D 窗口，可以看到一个旋转的三棱锥。细心的读者可能会发现，棱锥看上去好像有点透明，这是因为没有打开 Z 缓存的缘故，此时 Direct3D 只是简单地按图元格式，顺序渲染三角形，没有考虑平面之间的遮挡关系，从而导致问题的出现。

### 5.4 打开 Z 缓存

在 Direct3D 中，使用深度缓存区（Depth Buffer）来进行消隐处理（隐藏面消除），以确保实体被遮挡的部分不被显示。Z 缓存是最常用的一种深度缓存，它因为用 Z 坐标作为判断深度（远近）的依据而得名，其工作原理如图 14 所示，图中的渲染表面相当于 Direct3D 窗口，Z 缓存用来保存窗口中各个像素的深度。在消隐时，Direct3D 先用背景色（或纹理）填充渲染表面，Z 缓存则统一设置成最大深度，即投影变换中后裁剪平面的距离，然后逐像素处理渲染表面：对于任意一个像素，Direct3D 逐一测试所有与该像素重叠的三角形，如果三角形中像素对应点的 Z 坐标小于 Z 缓存中的数值，也就是说，此三角形离观察者较近，则像素取该点的颜色，同时像素在 Z 缓存中的深度也设为该点的 Z 坐标，然后继续测试下一个三角形... ...

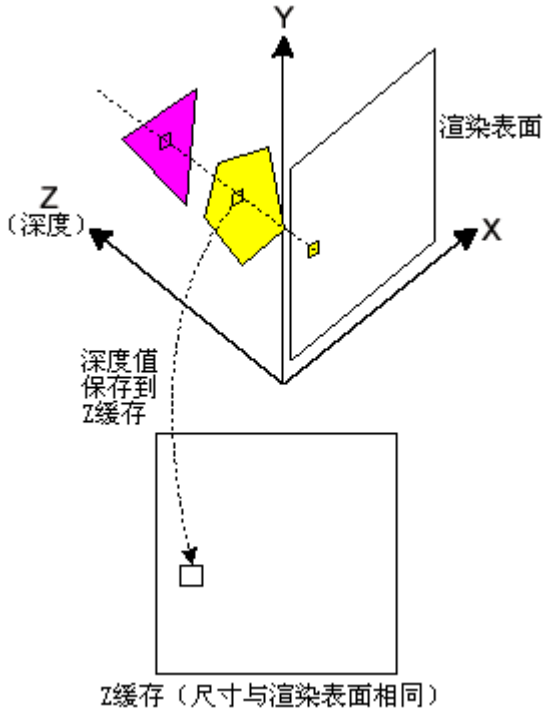


图 14

Z 缓存的工作原理说起来有些拗口，不过使用还是很容易的。首先在初始化函数 CD3DWnd::InitD3D 中添加如下代码，以便在创建设备对象的同时生成 Z 缓存：

（D3DWnd.cpp）

```
... ...
d3dpp.BackBufferFormat = D3DFMT_UNKNOWN;
d3dpp.EnableAutoDepthStencil = TRUE;
d3dpp.AutoDepthStencilFormat = D3DFMT_D16;  //生成 16 bit 的 Z 缓存
m_pD3D->CreateDevice( ... ...
//启用 Z 缓存，允许消隐处理
m_pDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
... ...
```

然后修改 CD3DWnd::Render 中 m_pDevice->Clear 的调用参数，在清除后备缓存区的同时，把 Z 缓存统一设置为最大深度 1.0：

（D3DWnd.cpp）

```
... ...
m_pDevice->Clear(
    0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
    D3DCOLOR_XRGB(0,0,255),
    1.0f, 0);
... ...
```

此时再编译运行程序，显示效果如图 15 所示。

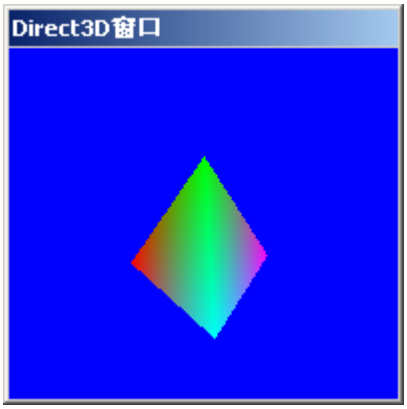


图 15

除 Z 缓存外，Direct3D 中还有一种深度缓存－W 缓存（W-Buffer），用于变换后的坐标空间（w 的全称是 Reciprocal Homogeneous W，简写为 RHW），限于篇幅，这里就不介绍了。

虽然本节使用了索引缓存绘制三棱锥，但它并不适合画多边形，因为索引缓存会产生公共顶点，其顶点法线不好确定。只不过该例程是使用顶点颜色进行渲染，没使用光照，因此不需要顶点法线。

## 6 画一个圆锥－灯光和材质

本节将通过画一个圆锥，介绍灯光（Light）和材质（Material）的用法，以及如何用多个图元构建实体。

### 6.1 基本概念

在前面的例程中，通过对顶点颜色进行插值来获取实体表面的颜色，这种简化的计算模型无法如实地反映真实世界。在自然界中，我们所看到的一切都是由光线产生的：光由光源出发，沿直线传播；当光线遇到物体时，一部分被吸收，剩余的被反射，该过程反复进行，直至光线能量耗尽，或者被人眼接收从而产生视觉。

在 Direct3D 中，用灯光和材质来模拟这个过程。灯光用于照亮实体，可分为环境光（Ambient Light）和直射光（Direct Light）：前者均匀充满整个场景，为所有实体提供一个恒定的照明，没有方向性；后者一般由光源产生，具有方向性。材质则定义了实体表面对光线的反射属性。

Direct3D 用结构 D3DCOLORVALUE 描述直射光和材质的颜色，它有 4 个浮点分量，分别代表红、绿、蓝、Alpha 混合，正常取值范围 0.0-1.0。其中 Alpha 混合用来产生透明效果，仅用于材质：0.0 表示完全透明；1.0 为不透明。

环境光颜色用一个 4 字节的整数 D3DCOLOR 描述，每个字节依次代表红、绿、蓝、Alpha 混合，取值范围 0-255，可以借助宏 D3DCOLOR_RGBA 来简化计算。和直射光一样，环境光也不使用 Alpha 混合。

虽然灯光和材质都有颜色，但其含义并不相同。灯光的颜色定义了光线中三原色的“数量”，红=绿=蓝=1.0 为白光，都取 0.0 表示没有光。材质的颜色代表了在光线发生反射时，三原色被反射的“数量”，红=绿=蓝=Alpha=1.0 表示所有光线都被反射，也就是说，材质看上去为白色，而红=绿=0.0，蓝=Alpha=1.0 则表示只有蓝光被反射，即材质为蓝色。

### 6.2 灯光

环境光的使用比较简单，Direct3D 把它作为一个渲染状态，通过调用 IDirect3DDevice9::SetRenderState 进行设置，对应的状态常数为 D3DRS_AMBIENT。以下着重介绍直射光的应用。

按光源划分，直射光可分为三种：

**1）点光源**

点光源（Point Light）从一个点向周围均匀地发射光线，如图 16 所示，家用的白炽灯就是一个点光源。点光源有颜色、位置、作用范围，光强随距离而衰减，没有方向（因为向全部方向发射）。

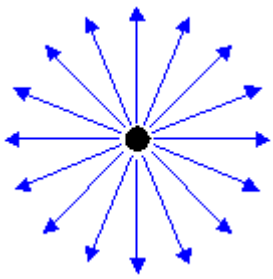

图 16

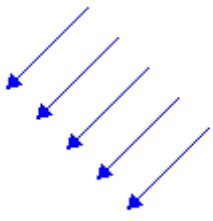

图 17

**2）平行光**

平行光（Directional Light）由相互平行的光线组成，如图 17 所示，最常见的例子就是阳光。平行光只有颜色和方向，没有位置，也没有作用范围和衰减，因此不论实体位于场景的何处，所受到的光照都相同。

**3）聚光灯**

聚光灯（Spotlight）是三种直射光中最复杂的一种，常见的例子有手电筒、探照灯。它的光束是一个圆锥，其截面如图 18 所示，分内、外核两部分：内核最亮，且亮度保持不变；外核较暗，沿径向有一个衰减。图 19 是聚光灯的示意图，其中夹角 Theta 和 Phi 定义了内、外核的大小。聚光灯有颜色、位置、方向（即光束中心所指方向）、作用范围、衰减（沿光线方向）。

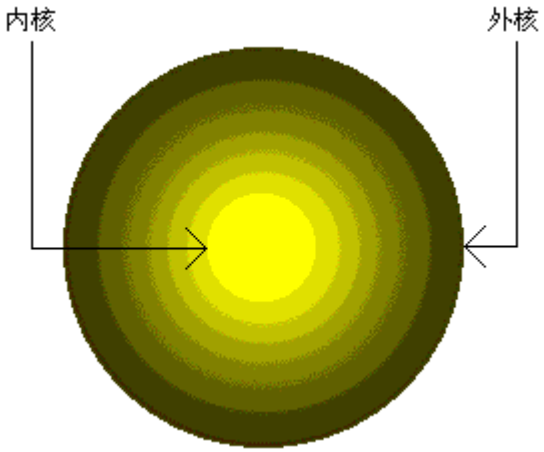


图 18

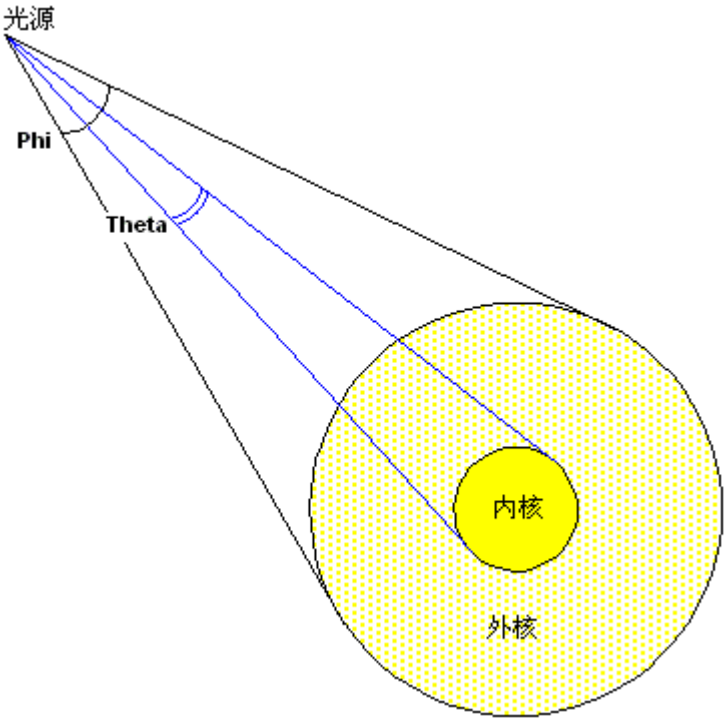


图 19

在 Direct3D 中，用结构 D3DLIGHT9 来描述直射光，它的定义如下：

```
typedef struct _D3DLIGHT9 {
    D3DLIGHTTYPE Type;          //类型：只能是点光源、平行光或聚光灯
    D3DCOLORVALUE Diffuse;      //灯光的漫反射颜色
    D3DCOLORVALUE Specular;     //灯光的镜面反射颜色
    D3DCOLORVALUE Ambient;      //灯光的环境光颜色
    D3DVECTOR Position;         //光源在世界坐标系的位置
    D3DVECTOR Direction;        //灯光的方向，建议使用单位矢量
    float Range;                //灯光的作用范围
    float Falloff;          //聚光灯内核到外核的衰减系数，通常取 1.0，表示均匀过渡
    float Attenuation0;         //距离衰减系数之一：通常取 0.0
    float Attenuation1;       //距离衰减系数之二：通常取一个大于 0 的常数
    float Attenuation2;         //距离衰减系数之三：通常取 0.0
    float Theta;                //聚光灯的内核大小
    float Phi;                  //聚光灯的外核大小
} D3DLIGHT9;
```

上述数据项中，最不好理解的恐怕要算灯光的颜色了，竟然有三种。在 Direct3D 的光照模型中，灯光效果由三部分组成：漫反射、镜面反射和环境光照。Direct3D 以灯光的漫反射颜色和材质的漫反射颜色为输入参数，计算最终的漫反射效果，镜面反射与此类似。而灯光的环境光颜色则参于计算整个场景的环境光照，此前以渲染状态方式设置的环境光相当于公式中的常数项。

以上只是一个粗略的解释，读者在编程时，不妨试着分别改变这三种颜色，看看每种颜色所起的作用。如果对 Direct3D 的光照模型感兴趣，推荐阅读 SDK 中“Mathematics of Lighting”这篇文章，其中给出了详细的计算公式。

设置好 D3DLIGHT9 的各个成员后，调用 IDirect3DDevice9::SetLight 把直射光加入场景，然后还要执行 IDirect3DDevice9::LightEnable 激活它。

使用灯光会增加渲染的计算量，按从小到大排序，依次为：环境光、平行光、点光源、聚光灯。因此在编程时，要少用聚光灯，多用平行光和点光源。

### 6.3 材质

前面已经提过，材质用于描述实体的反光性能，Direct3D 使用结构 D3DMATERIAL9 保存材质，它有如下成员：

```
typedef struct _D3DMATERIAL9 {
    D3DCOLORVALUE Diffuse;      //材质的漫反射颜色
    D3DCOLORVALUE Ambient;      //材质的环境光颜色
    D3DCOLORVALUE Specular;     //材质的镜面反射颜色
    D3DCOLORVALUE Emissive;     //材质的发射颜色
    float Power;                //材质的镜面反射强度
} D3DMATERIAL9;
```

漫反射颜色定义了材质对灯光中漫反射分量的反射性能，环境光颜色定义了材质对环境光照的反射性能，这二者结合在一起，决定了实体的外观颜色。在编程中，它们通常取相同的值。

镜面反射颜色定义了材质对灯光中镜面反射分量的反射性能，一般用于在材质表面产生高光部分，使实体看上去有光泽，通常取白或亮灰。Power 决定镜面反射的强度，或者说，实体的光滑程度。Power 越大，反光越强。图 20 是效果对比图，其中左边使用了镜面反射，

Power 为 10。

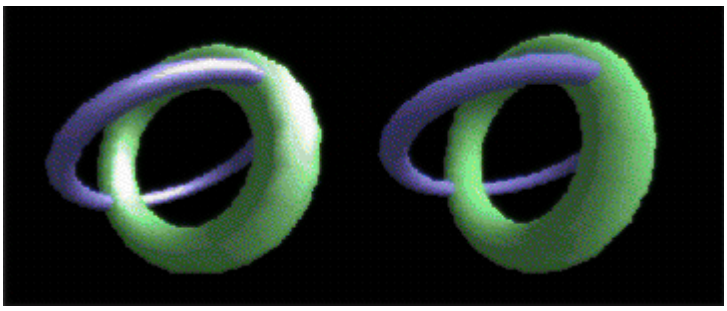

图 20

发射颜色用于定义自身可以发光的材质，这种光只是让实体看上去更明亮，不能用来照明。

在 D3DMATERIAL9 中设置好材质的各项属性后，调用 IDirect3DDevice9::SetMaterial 把它加入场景。

### 6.4 画一个圆锥

为了便于读者理解，本节及后续章节的例程均不使用索引缓存，因此我们将在第 4 节例程的基础上进行修改。不过在输入下列代码之前，请先参照第 5 节的示例，打开 Z 缓存。

#### 6.4.1 建模

圆锥由一个曲面和一个底面构成，如图 21 所示，用两个图元构建实体比较合理，其中每个图元各代表一个面。考虑到圆锥曲面展开后为一个扇面，所以图元格式采用三角扇形，而底面是一个圆，也采用三角扇形，取圆锥外表面做图元的正面。在图 21 中，A 为圆锥的顶点，B 为底面的圆心，把底面分成 30 等份，得到 30 个分割点 $C_1$、$C_2$ ... ... $C_{30}$，则圆锥曲面的图元所对应的顶点序列为 A、$C_1$、$C_2$ ... ... $C_{30}$、$C_1$，共 32 个顶点，注意在结尾处 $C_1$ 点重复出现了一次，否则画出的圆锥不完整。同理底面也由 32 个顶点组成，依次为 B、$C_1$、$C_{30}$、$C_{29}$ ... ... $C_1$。

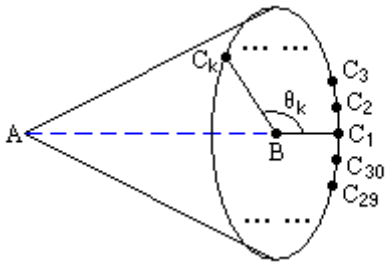


图 21

设圆锥高为 2，底面半径为 1，以圆锥中心线 AB（图中的蓝线）做 X 轴，坐标原点设在 AB 中点，A 的坐标为（-1，0，0），则 B 的坐标为（1，0，0），$C_1$ 至 $C_{30}$ 的坐标为（1，$\sin\theta_k$，$\cos\theta_k$）。

由于使用灯光照明，还需要提供顶点法线：对于圆锥曲面，A 的顶点法线取 X 轴的负方向，即矢量 {-1，0，0}，$C_1$ 至 $C_{30}$ 的顶点法线取有向线段 $BC_k$，矢量为 {0，$\sin\theta_k$，$\sin\theta_k$}；对于圆锥底面，顶点法线都设为 X 轴的正方向，矢量为 {1，0，0}。请注意，在 Direct3D 中，要求顶点法线为单位矢量。

现在开始输入代码，首先定义两个图元的 FVF，因为使用灯光渲染，顶点颜色不再需要，新的 FVF 包括坐标和顶点法线：

（D3DWnd.cpp）

```
... ...
#include "D3DWnd.h"
//圆锥曲面的 FVF 格式：坐标、顶点法线
struct CUSTOMVERTEX1
{
    D3DXVECTOR3 position;    //顶点坐标
    D3DXVECTOR3 normal;      //顶点法线
};
#define D3DFVF_CUSTOMVERTEX1 (D3DFVF_XYZ | D3DFVF_NORMAL)
//圆锥底面的 FVF 格式：坐标、顶点法线
struct CUSTOMVERTEX2
{
    D3DXVECTOR3 position;    //顶点坐标
    D3DXVECTOR3 normal;      //顶点法线
};
#define D3DFVF_CUSTOMVERTEX2 (D3DFVF_XYZ | D3DFVF_NORMAL)
... ...
```

为 CD3DWnd 增加两个数据成员 m_pVB1 和 m_pVB2，分别保存圆锥曲面和底面的顶点缓存区接口指针，原有的 m_pVB 予以删除：

（D3DWnd.h）

```
... ...
void SetupMatrices();

LPDIRECT3DVERTEXBUFFER9 m_pVB1;  //圆锥曲面的顶点缓存区接口指针
LPDIRECT3DVERTEXBUFFER9 m_pVB2;  //圆锥底面的顶点缓存区接口指针
... ...
```

修改 CD3DWnd::InitGeometry，建立圆锥的几何模型：

（D3DWnd.cpp）

```
void CD3DWnd::InitGeometry()
{
    //建立圆锥曲面的数学模型
    CUSTOMVERTEX1 vertices1[32];
    vertices1[0].position = D3DXVECTOR3( -1.0f, 0.0f, 0.0f ); //点 A 的坐标
    vertices1[0].normal = D3DXVECTOR3( -1.0f, 0.0f, 0.0f ); //点 A 的法线矢量
    for (int i = 1; i < 32; i++)
    {    //计算顶点序列 C1、C2 ... ... C30、C1 的坐标和法线
        float theta = (i-1)*12*D3DX_PI/180;
        vertices1[i].position = D3DXVECTOR3( 1.0f, sin(theta), cos(theta) );
        vertices1[i].normal = D3DXVECTOR3( 0.0f, sin(theta), cos(theta) );
```

```
    }
    //创建圆锥曲面的顶点缓存区，填入顶点数据
    m_pDevice->CreateVertexBuffer(
        sizeof(vertices1), 0, D3DFVF_CUSTOMVERTEX1,
        D3DPOOL_DEFAULT, &m_pVB1, NULL );
    void* pVertices;
    m_pVB1->Lock( 0, sizeof(vertices1), (void**)&pVertices, 0 );
    memcpy( pVertices, vertices1, sizeof(vertices1) );
    m_pVB1->Unlock();

    //建立圆锥底面的数学模型
    CUSTOMVERTEX2 vertices2[32];
    vertices2[0].position = D3DXVECTOR3( 1.0f, 0.0f, 0.0f ); //点 B 的坐标
    vertices2[0].normal = D3DXVECTOR3( 1.0f, 0.0f, 0.0f ); //点 B 的法线矢量
    for (i = 1; i < 32; i++)
    {   //计算顶点序列 C1、C30、C29 ... ... C1 的坐标和法线
        vertices2[i].position = vertices1[32-i].position;
        vertices2[i].normal = D3DXVECTOR3( 1.0f, 0.0f, 0.0f );
    }
    //创建圆锥底面的顶点缓存区，填入顶点数据
    m_pDevice->CreateVertexBuffer(
        sizeof(vertices2), 0, D3DFVF_CUSTOMVERTEX2,
        D3DPOOL_DEFAULT, &m_pVB2, NULL );
    m_pVB2->Lock( 0, sizeof(vertices2), (void**)&pVertices, 0 );
    memcpy( pVertices, vertices2, sizeof(vertices2) );
    m_pVB2->Unlock();
}
```

上述代码为每个图元定义了各自的 FVF 和顶点缓存区，其实如果顶点格式相同，完全可以把所有顶点放到同一个缓存区，然后在调用 DrawPrimitive 时，给出所绘图元的第一个顶点在缓存区中的偏移量即可。现在之所以分开存储，是出于对后续章节的考虑，在下一节中，两个图元将使用不同的顶点格式。

修改函数 CD3DWnd::Cleanup，释放新增加的接口指针，代码如下：

（D3DWnd.cpp）

```
void CD3DWnd::Cleanup()
{
    m_pVB1->Release();          //释放圆锥曲面的顶点缓存区
    m_pVB2->Release();          //释放圆锥底面的顶点缓存区
    m_pDevice->Release();       //释放设备对象
    m_pD3D->Release();          //释放 Direct3D 对象
}
```

在初始化函数 CD3DWnd::InitD3D 中设置渲染状态 D3DRS_NORMALIZENORMALS 为 TRUE，确保顶点法线总是单位矢量，不受坐标变换的影响，这有助于提高渲染的精确性，

但会增加处理器负担。以前的程序为了使用顶点颜色渲染，把光照处理禁用了，现在要打开。另外，由于这一次的实体是一个封闭图形，不需要渲染背面，因此要把原来允许渲染背面的语句删掉：

（D3DWnd.cpp）

```
    ... ...
    m_pD3D->CreateDevice( ... ...
    m_pDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
    //打开光照处理
    m_pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
    //自动对法线矢量进行归一化处理
    m_pDevice->SetRenderState( D3DRS_NORMALIZENORMALS, TRUE );
}
```

### 6.4.2 添加灯光和材质

我们使用一个白色的平行光进行照明，灯光方向指向左下方，矢量为｛-1，-1，0｝；环境光设为一个亮度很低的灰度光；圆锥曲面的材质设为白色，底面设为黄色，没有镜面反射。

为 CD3DWnd 添加三个成员函数：SetLight、SetMaterial1、SetMaterial2，分别用于设置灯光、圆锥曲面材质、圆锥底面材质：

（D3DWnd.h）

```
... ...
LPDIRECT3DVERTEXBUFFER9 m_pVB2;
void SetLight();          //该函数用于设置灯光
void SetMaterial1();      //该函数用于设置圆锥曲面的材质
void SetMaterial2();      //该函数用于设置圆锥底面的材质
... ...
```

（D3DWnd.cpp）

```
void CD3DWnd::SetLight()
{
    //创建一个白色的平行光
    D3DLIGHT9 light;
    ::ZeroMemory( &light, sizeof(D3DLIGHT9) );
    light.Type = D3DLIGHT_DIRECTIONAL;     //灯光类型
    light.Diffuse.r = 1.0f;
    light.Diffuse.g = 1.0f;
    light.Diffuse.b = 1.0f;
    light.Direction = D3DXVECTOR3( -1.0f, -1.0f, 0.0f );
    light.Range    = 1000.0f;                  //灯光的作用范围
    m_pDevice->SetLight( 0, &light );     //设置灯光，参数 1 为灯光的索引号
    m_pDevice->LightEnable( 0, TRUE );//打开灯光，参数 1 为灯光的索引号
    //设置环境光
    m_pDevice->SetRenderState( D3DRS_AMBIENT, D3DCOLOR_RGBA(32,32,32,0) );
}
```

```
void CD3DWnd::SetMaterial1()
{
    //创建一个白色的材质
    D3DMATERIAL9 mtrl;
    ::ZeroMemory( &mtrl, sizeof(D3DMATERIAL9) );
    mtrl.Diffuse.r = mtrl.Ambient.r = 1.0f;
    mtrl.Diffuse.g = mtrl.Ambient.g = 1.0f;
    mtrl.Diffuse.b = mtrl.Ambient.b = 1.0f;
    mtrl.Diffuse.a = mtrl.Ambient.a = 1.0f;
    m_pDevice->SetMaterial( &mtrl );    //设置材质
}

void CD3DWnd::SetMaterial2()
{
    //创建一个黄色的材质
    D3DMATERIAL9 mtrl;
    ::ZeroMemory( &mtrl, sizeof(D3DMATERIAL9) );
    mtrl.Diffuse.r = mtrl.Ambient.r = 1.0f;
    mtrl.Diffuse.g = mtrl.Ambient.g = 1.0f;
    mtrl.Diffuse.b = mtrl.Ambient.b = 0.0f;
    mtrl.Diffuse.a = mtrl.Ambient.a = 1.0f;
    m_pDevice->SetMaterial( &mtrl );    //设置材质
}
```

修改渲染函数 CD3DWnd::Render，加入圆锥的绘制代码：

（D3DWnd.cpp）

```
... ...
m_pDevice->BeginScene();
SetupMatrices();    //设置变换矩阵
SetLight();    //设置灯光
//绘制圆锥曲面的图元
SetMaterial1();    //使用白色的材质
m_pDevice->SetFVF( D3DFVF_CUSTOMVERTEX1 );
m_pDevice->SetStreamSource( 0, m_pVB1, 0, sizeof(CUSTOMVERTEX1) );
m_pDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 30 );
//绘制圆锥底面的图元
SetMaterial2();    //使用黄色的材质
m_pDevice->SetFVF( D3DFVF_CUSTOMVERTEX2 );
m_pDevice->SetStreamSource( 0, m_pVB2, 0, sizeof(CUSTOMVERTEX2) );
m_pDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 30 );
m_pDevice->EndScene();
... ...
```

编译运行程序，效果如图 22 所示。如果显示不正确，看是不是忘了打开 Z 缓存。

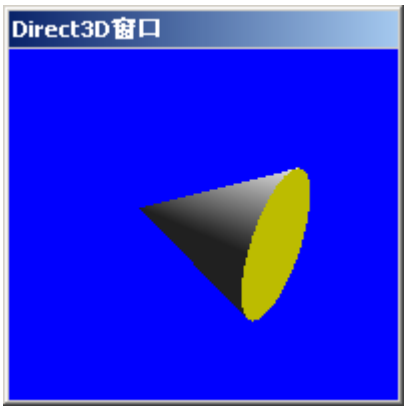


图 22

以上所画的圆锥，在由多个图元构成的实体中，只能算是一个比较简单的例子。绘制多图元实体的一般方法是：首先为图元设置各自的世界变换矩阵、材质和纹理（另外两个变换矩阵和灯光都属于全局性参数，设置一次即可）；然后对该图元调用 IDirect3DDevice9::DrawPrimitive 方法。

### 6.5 高洛德着色和平面着色

在本节的最后，简要介绍一下 Direct3D 中用于渲染三角形的两种着色算法：

- 平面着色（Flat Shading），也叫做“恒量着色”，是一种最简单也是最快速的着色算法。在此算法中，每个三角形使用一种颜色进行填充。这种方法的显示效果最差，一般用在要求速度重于细节的场合，如生成预览；
- 高洛德着色（Gouraud Shading），又称高式着色，其显示效果要好得多，它是游戏中使用最广泛的一种着色算法。高洛德着色对实体模型各顶点的颜色进行平滑、融合处理，为每个三角形上的每个点赋以一个独立的颜色值，将三角形着上较为顺滑的渐变色，使其外观更加真实，但着色速度比平面着色慢许多。

Direct3D 缺省使用高洛德着色，如果想改用平面着色，只需调用 IDirect3DDevice9::SetRenderState，把渲染状态 D3DRS_SHADEMODE 设置为 D3DSHADE_FLAT 即可。图 23 是这两种算法的效果对比，其中左边使用了平面着色。

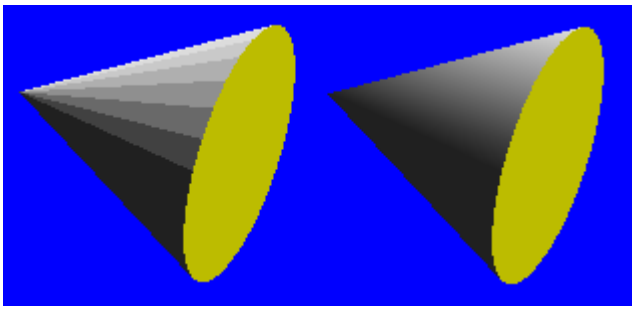

图 23

## 7 为圆锥添加纹理

本节将通过为上一节所画的圆锥曲面添加纹理，演示纹理的基本用法。

### 7.1 基本概念

只有材质的实体看上去就象塑料制品，还不足以反映我们这个五彩缤纷的世界，为此 Direct3D 引入了纹理（Texture）技术。纹理也就是通常所说的贴图，它通过在三维的模型表面覆盖上二维的图片，使实体更具有真实感，比如在家具表面贴上木纹，或者把草、泥土和岩石等图片贴在构成山的图元表面，以得到一个真实的山坡。图 24 中，左边是使用纹理的战斗机模型，右边没使用纹理，显然左图更真实一些。Direct3D 支持多层纹理，最高可达 8 层。

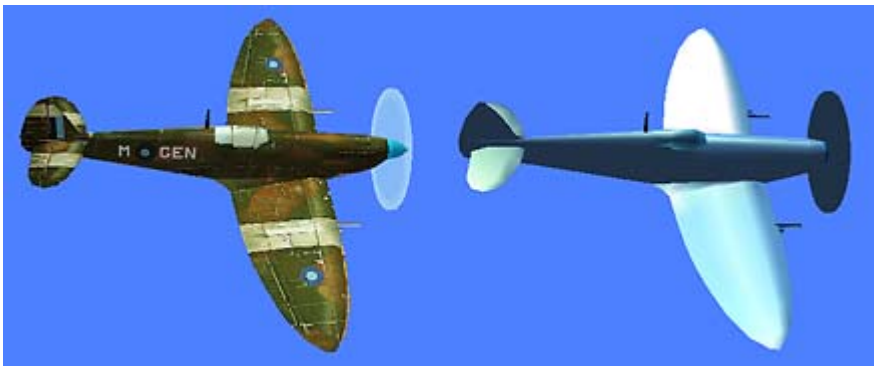

图 24

纹理都是一些标准的位图，支持 BMP、JPG、PNG、TGA 等格式。虽然 Direct3D 对纹理图片的大小没有限制，但为了程序的执行效率，最好使用正方形图片，而且边长是 2 的 n 次方，比如 64 x 64、128 x 128、256 x 256 等等。

在开始使用纹理之前，读者有必要了解一些名词：

**1）纹理坐标**

纹理图片本身构成了一个二维的坐标空间，纹理坐标（Texture Coordinate）用于在纹理上指定一个点，如图 25 所示，其中 u 为横坐标，v 为纵坐标。不论纹理的大小如何，其左上角总是（0，0），右下角总是（1，1），易知中心点的纹理坐标为（0.5，0.5）。

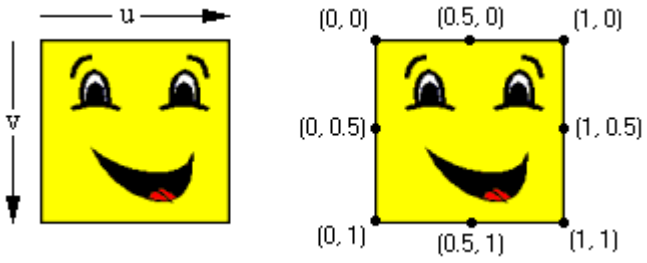


图 25

当把纹理应用于图元时，需要为每个顶点指定一组纹理坐标，标明该顶点在贴图中的位置，从而建立起图元和纹理图片之间的映射关系。我们不妨把纹理图片想象成一片弹性很好的橡皮薄膜，贴图过程就相当于用钉子把橡皮固定在与其纹理坐标相对应的顶点上。

纹理坐标的正常取值范围为 0-1，但 Direct3D 也允许纹理坐标取此范围之外的值，以获取某些特殊的纹理效果。图 26 是一个纹理重复的例子，进一步的讨论参见 SDK 文章‘ Texture

Addressing Modes ”。

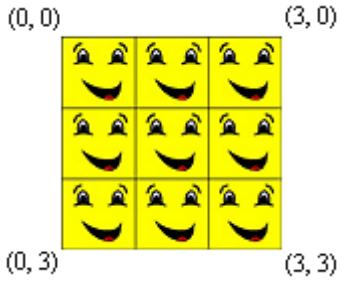


图 26

**2）纹理滤波**

纹理滤波（Texture Filtering）是指纹理图片映射到图元的方式，Direct3D 支持以下方式：

- 最近点采样
- 双线性滤波
- 各向异性滤波
- Mipmap 滤波

其中最近点采样是 Direct3D 中的缺省方式，速度快、但效果一般，可以通过 IDirect3DDevice9::SetSamplerState 来改变滤波方式。关于各种方式的优劣请参见 SDK 文章“ Texture Filtering ”，下面的例子把第一层纹理的滤波方式设为双线性滤波：

```
pdevice->SetSamplerState( 0, D3DSAMP_MAGFILTER, D3DTEXF_LINEAR  );
pdevice->SetSamplerState( 0, D3DSAMP_MINFILTER, D3DTEXF_LINEAR  );
pdevice->SetSamplerState( 0, D3DSAMP_MIPFILTER, D3DTEXF_LINEAR  );
```

**3）纹理混合**

当使用纹理时，实体的外观由纹理图片和材质经过某种计算获得，这个过程被称作纹理混合（Texture Blending），详细情况参见 SDK 的文章“ Texture Blending ”，下面的例子把第一层纹理的混合方式设为“ 纹理 ＋ 材质的漫反射 ”：

```
pdevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_ADD);        //运算类型：加
pdevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE); //运算对象 1：纹理
pdevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE); //运算对象 2：材质
```

## 7.2 创建纹理

现在来为上一节所画圆锥的曲面添加纹理。在 Direct3D 中，纹理属于 COM 对象，既可以从现有的图片文件中直接创建，也可以先创建一个空的纹理对象，然后再进行填充。出于方便，这里选用前一种方式。

打开上一节例程，为 CD3DWnd 添加一个数据成员，用来保存纹理对象的接口指针：

（D3DWnd.h）

```
... ...
void SetMaterial2();

LPDIRECT3DTEXTURE9 m_pTexture;        //纹理对象的接口指针
... ...
```

修改函数 CD3DWnd::InitGeometry，添加纹理对象的创建代码，以下假设纹理图片的文

件名为 texture.jpg，和源程序位于同一目录：

（D3DWnd.cpp）

```
void CD3DWnd::InitGeometry()
{
    //从图片文件中直接创建纹理对象
    ::D3DXCreateTextureFromFile( m_pDevice, "texture.jpg", &m_pTexture );
    ... ...
```

在函数 CD3DWnd::Cleanup 中添加纹理对象的释放代码：

（D3DWnd.cpp）

```
... ...
m_pVB2->Release();
m_pTexture->Release();      //释放纹理对象
m_pDevice->Release();
... ...
```

**7.3 用纹理渲染**

首先要确定图元各顶点的纹理坐标，圆锥曲面展开后是一个扇形，它与纹理图片的对应关系如图 27 所示，其中 $C_1$、$C_2$ ... ... $C_{30}$ 是等分点，则 A 的纹理坐标为（0，0.5），$C_k$ 的纹理坐标为（$0.5\sin\beta_k$ ，$0.5-0.5\cos\beta_k$）。

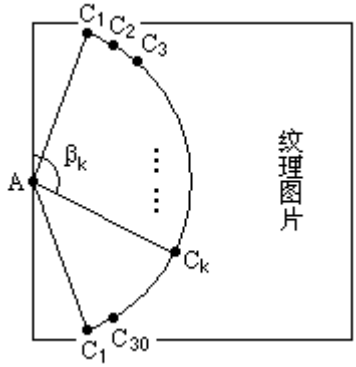


图 27

修改圆锥曲面的 FVF 定义，加入纹理坐标：

（D3DWnd.cpp）

```
... ...
#include "D3DWnd.h"
//圆锥曲面的 FVF 格式：坐标、顶点法线、纹理坐标
struct CUSTOMVERTEX1
{
    D3DXVECTOR3 position;     //顶点坐标
    D3DXVECTOR3 normal;       //顶点法线
    float tu, tv;             //纹理坐标
};
//D3DFVF_TEX1 表示只使用一层纹理
```

```
#define D3DFVF_CUSTOMVERTEX1 (D3DFVF_XYZ | D3DFVF_NORMAL | D3DFVF_TEX1)
... ...
```

然后在函数 CD3DWnd::InitGeometry 中计算曲面图元各顶点的纹理坐标：

（D3DWnd.cpp）

```
... ...
//建立圆锥曲面的数学模型
CUSTOMVERTEX1 vertices1[32];
vertices1[0].position = D3DXVECTOR3( -1.0f, 0.0f, 0.0f );
vertices1[0].normal = D3DXVECTOR3( -1.0f, 0.0f, 0.0f );
vertices1[0].tu = 0.0f;          //点 A 的纹理坐标
vertices1[0].tv = 0.5f;
for (int i = 1; i < 32; i++)
{
    float theta = (i-1)*12*D3DX_PI/180;
    vertices1[i].position = D3DXVECTOR3( 1.0f, sin(theta), cos(theta) );
    vertices1[i].normal = D3DXVECTOR3( 0.0f, sin(theta), cos(theta) );
    //计算顶点序列的纹理坐标
    float m = sqrt(5);
    float beta = (0.5 – 1.0/m + (i-1.0)/15/m)*D3DX_PI;
    vertices1[i].tu = 0.5*sin(beta);
    vertices1[i].tv = 0.5 - 0.5*cos(beta);
}
... ...
```

最后修改函数 CD3DWnd::Render，把纹理加入场景，并设置纹理的混合方式：

（D3DWnd.cpp）

```
... ...
//绘制圆锥曲面的图元
SetMaterial1();
//把纹理加入场景，其中参数 1 为纹理的索引号，取值范围 0-7，
//分别代表第一至第八层纹理。这里只用了一层纹理，因此索引号为 0。
m_pDevice->SetTexture( 0, m_pTexture );
//设置纹理混合方式为“ 纹理 × 材质的漫反射 ”。
//其实可以不用设置，因为这是 Direct3D 缺省的混合方式。
m_pDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_MODULATE );
m_pDevice->SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TEXTURE );
m_pDevice->SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_DIFFUSE );
m_pDevice->SetFVF( D3DFVF_CUSTOMVERTEX1 );
m_pDevice->SetStreamSource( 0, m_pVB1, 0, sizeof(CUSTOMVERTEX1) );
m_pDevice->DrawPrimitive( D3DPT_TRIANGLEFAN, 0, 30 );
//纹理用完后，应该从场景中移除
m_pDevice->SetTexture( 0, NULL );
m_pDevice->SetTextureStageState( 0, D3DTSS_COLOROP, D3DTOP_DISABLE );
```

```
//绘制圆锥底面的图元
SetMaterial2();
... ...
```

编译运行程序，注意要从 VC6 的集成环境中运行，以确保当前目录为源程序目录，否则纹理文件无法打开。运行效果如图 28 所示，图 29 是纹理图片 texture.jpg 的内容，读者可对比一下，看看它是如何映射到圆锥的。

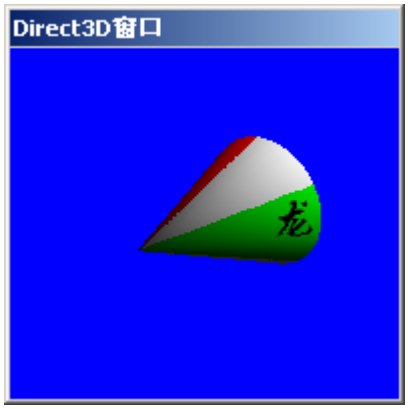


图 28

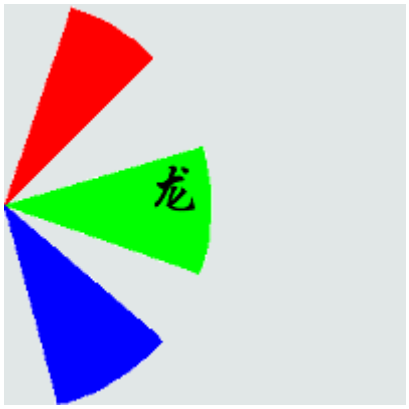

图 29

# 8 Mesh 模型

## 8.1 什么是 Mesh 模型

在前面的例程中，都是采用手工方法推导实体的几何模型，对付简单的应用还行，如果场景复杂，涉及的实体较多，工作量就太大了。因此实际工作中，常常使用 3DS MAX、Maya、MultiGen 等三维软件进行辅助设计，通过这些软件生成的实体模型不仅包括顶点的几何数据，还包括了材质和纹理，习惯上把这些模型称之为 Mesh。

Mesh 模型有多种格式，如 3DS MAX 使用的.max、.3ds、MultiGen 使用的.flt 等等。Direct3D 内置了对.x 格式的模型的支持，该模型由一个后缀为.x 的数据文件和若干个纹理图片文件组成。虽然很少会有商业游戏直接使用.x 格式，不过拿它入门还是足够了。以下所说的 Mesh 模型均指.x 格式。

在 Direct3D 中，一个 Mesh 模型通常由成百上千个三角形构成，这些三角形被划分成若干个子集，每个子集拥有自己的材质和纹理。Mesh 模型属于 COM 对象，通过它的接口 ID3DXMesh 绘制 Mesh 非常简单：首先创建 Mesh 对象，然后顺序为各个子集设置各自的材质和纹理，并调用 ID3DXMesh::DrawSubset 绘制该子集。

在创建 Mesh 对象时，除了接口 ID3DXMesh 外，还会得到一个接口 ID3DXBuffer，我们将通过它获取模型中各个子集的材质和纹理（准确地说，应该是纹理的图片文件名）。

由于.x 是微软自己搞的一套格式，模型文件比较少见，为此微软提供了两个工具 Conv3ds.exe 和 XSkinExp.dle，分别用于把.3ds 和.max 转换成.x 格式。这两个工具位于 SDK 的扩展包“ DirectX 9.0 SDK Extras: Direct3D ”，需要另外下载。

## 8.2 绘制 Mesh 模型

下面介绍如何在场景中加载 Mesh 模型，我们将以上一节的例程为基础进行修改。

由于 Mesh 对象有自己的顶点缓存区、材质和纹理，原先的代码不再有用，予以删除，包括：

- CD3DWnd 的数据成员 m_pVB1、m_pVB2、m_pTexture；
- 顶点的 FVF 定义；
- 函数 CD3DWnd::InitGeometry 中的所有语句，只保留一个框架；
- 函数 CD3DWnd::Cleanup 中释放 m_pVB1、m_pVB2、m_pTexture 的语句；
- CD3DWnd 的成员函数 SetMaterial1 和 SetMaterial2。

接下来用.x 文件创建 Mesh 对象，首先为 CD3DWnd 添加下列数据成员，用于保存 Mesh 对象的接口指针以及模型中各个子集的材质和纹理：

（D3DWnd.h）

```
... ...
void SetLight();

LPD3DXMESH m_pMesh;        //Mesh 对象的接口指针
D3DMATERIAL9 *m_pMeshMaterials;  //用于保存模型中各个子集的材质
LPDIRECT3DTEXTURE9 *m_pMeshTextures;//用于保存模型中各个子集的纹理
DWORD m_dwNumSubsets;      //模型中子集的数目
... ...
```

在函数 CD3DWnd::InitGeometry 中创建 Mesh 对象，并获取材质和纹理。以下假设模型

的.x 文件名为 su37.x，和纹理图片一起位于源程序所在目录：

（D3DWnd.cpp）

```
void CD3DWnd::InitGeometry()
{
    //我们将通过该接口指针访问 Mesh 对象的材质和纹理
    LPD3DXBUFFER pD3DXMtrlBuffer;
    //从.x 文件创建 Mesh 对象
    ::D3DXLoadMeshFromX(
        "su37.x", D3DXMESH_SYSTEMMEM, m_pDevice,
        NULL, &pD3DXMtrlBuffer, NULL,
        &m_dwNumSubsets,   //返回模型中子集的数目
        &m_pMesh );        //返回 Mesh 对象的接口指针
    m_pMeshMaterials = new D3DMATERIAL9[m_dwNumSubsets];
    m_pMeshTextures = new LPDIRECT3DTEXTURE9[m_dwNumSubsets];
    //从接口指针 pD3DXMtrlBuffer 获得各个子集的材质和纹理图片的文件名
    D3DXMATERIAL* d3dxMaterials =
        (D3DXMATERIAL*)pD3DXMtrlBuffer->GetBufferPointer();
    for( DWORD i=0; i < m_dwNumSubsets; i++ )
    {
        //复制子集的材质
        m_pMeshMaterials[i] = d3dxMaterials[i].MatD3D;
        //Direct3D 在调入 Mesh 模型时，没有设置材质的环境光颜色，
        //这里把它设置成和漫反射一样的颜色。
        m_pMeshMaterials[i].Ambient = m_pMeshMaterials[i].Diffuse;
        //调入纹理图片
        ::D3DXCreateTextureFromFile(
            m_pDevice,
            d3dxMaterials[i].pTextureFilename,  //纹理图片的文件名
            &m_pMeshTextures[i] );
    }
    pD3DXMtrlBuffer->Release();   //释放接口
}
```

在 CD3DWnd::Cleanup 中释放新创建的 COM 对象：

（D3DWnd.cpp）

```
... ...
for (DWORD i = 0; i < m_dwNumSubsets; i++)
    m_pMeshTextures[i]->Release();      //释放纹理对象
delete [] m_pMeshMaterials;
delete [] m_pMeshTextures;
m_pMesh->Release();         //释放 Mesh 对象
m_pDevice->Release();
... ...
```

由于模型比较大，如果按原尺寸显示，在 Direct3D 窗口中只能看到部分画面，因此要在世界变换矩阵中增加一个缩放矩阵，对其进行缩小。函数 CD3DWnd::SetupMatrices 修改如下：

（D3DWnd.cpp）

```
... ...
float angle = m_nRotateY * D3DX_PI / 180;
D3DXMATRIX matWorld;
D3DXMATRIX matRotate;
D3DXMATRIX matZoom;
//计算旋转变换矩阵
::D3DXMatrixRotationY( &matRotate, angle );
//计算缩放变换矩阵：缩小 5 倍
::D3DXMatrixScaling( &matZoom, 0.2f, 0.2f, 0.2f );
//世界变换矩阵＝缩放矩阵×旋转矩阵
::D3DXMatrixMultiply( &matWorld, &matZoom, &matRotate );
//把世界变换矩阵设置到渲染环境
m_pDevice->SetTransform( D3DTS_WORLD, &matWorld );
... ...
```

最后修改函数 CD3DWnd::Render，渲染 Mesh 模型：

（D3DWnd.cpp）

```
... ...
m_pDevice->BeginScene();
SetupMatrices();
SetLight();
for( DWORD i=0; i < m_dwNumSubsets; i++ )
{
    //设置子集的材质
    m_pDevice->SetMaterial( &m_pMeshMaterials[i] );
    //设置子集的纹理，混合方式使用缺省值：纹理 × 材质的漫反射
    m_pDevice->SetTexture( 0, m_pMeshTextures[i] );
    //绘制子集
    m_pMesh->DrawSubset( i );
    m_pDevice->SetTexture( 0, NULL);
}
m_pDevice->EndScene();
... ...
```

编译程序，然后从 VC6 的集成环境中运行，以确保当前目录为源程序目录，运行结果如图 30 所示。如果觉得画面太暗，可以把环境光设置成白色（红=绿=蓝=255）。


图 30

## 9 显示文本

Direct3D 中的文本可分为二维和三维两种文本。

从本质上讲，二维文本的显示是借助纹理实现的。Direct3D 把这些实现细节封装在字体接口 ID3DXFont，通过调用其方法 DrawText，可以很方便地在窗口中输出文本。DrawText 使用窗口坐标系，下面的例子在窗口的左上方显示一行文本：

```
CFont font;
font.CreatePointFont( 120, "宋体", NULL );       //创建显示文本所用的字体
LPD3DXFONT pfont;
::D3DXCreateFont( pdevice, (HFONT)font.m_hObject, &pfont ); //从已有字体创建接口
pfont->DrawText(                //调用方法 ID3DXFont::DrawText 显示文本
                "二维文本",   //文本内容
                8,                //文本长度
                CRect(0,0,100,50), //文本的显示区域，使用窗口坐标
                DT_LEFT,          //显示格式：左对齐
                D3DCOLOR_XRGB(255,0,0) );   //文本颜色：红色
```

注意，ID3DXFont::DrawText 的调用必须放在 IDirect3DDevice9::BeginScene 和 IDirect3DDevice9::EndScene 之间进行。

三维文本是通过 Mesh 模型实现的。Direct3D 提供了一个函数 D3DXCreateText，可以生成文本的 Mesh 模型，然后再用上一节介绍的方法显示。文本 Mesh 模型只有一个子集，而且不包括材质和纹理，需要另外定义。另外，D3DXCreateText 不支持汉字。

现在我们来为上一节的例程添加文本显示功能，首先删除下列无用的代码：

- CD3DWnd 的数据成员 m_pMeshMaterials、m_pMeshTextures 和 m_dwNumSubsets；
- 函数 CD3DWnd::InitGeometry 中的所有语句，只保留一个框架；
- 函数 CD3DWnd::Cleanup 中释放 m_pMeshMaterials 和 m_pMeshTextures 的语句；

为 CD3DWnd 添加一个数据成员，用来保存二维文本的字体接口指针：

（D3DWnd.h）

```
... ...
LPD3DXMESH m_pMesh;
LPD3DXFONT m_p2DFont;     //用于显示二维文本的字体接口指针
... ...
```

在 CD3DWnd::InitGeometry 中分别创建二维文本的字体接口和三维文本的 Mesh 对象：

（D3DWnd.cpp）

```
void CD3DWnd::InitGeometry()
{
    //创建二维文本所用的字体：宋体
    CFont font1;
    font1.CreatePointFont( 120, "宋体", NULL );
    //创建二维文本的字体接口
    ::D3DXCreateFont( m_pDevice, (HFONT)font1.m_hObject, &m_p2DFont );

    //创建三维文本所用的字体：新罗马（必须使用 True Type 字体）
```

```
        CFont font2;
        font2.CreatePointFont( 100, "Times New Roman", NULL );
        //创建好的字体对象不能直接传递给函数 D3DXCreateText，必须
        //先选入一个设备环境，然后把设备环境句柄作为参数传递给函数。
        CDC memdc;
        memdc.CreateCompatibleDC( NULL );
        memdc.SelectObject( &font2 );
        //创建三维文本的 Mesh 对象
        ::D3DXCreateText(
            m_pDevice,
            memdc.m_hDC, //Direct3D 使用设备环境中的字体来创建 Mesh 对象
            "3D",    //文本内容
            0.001f,  //定义了字体轮廓的圆滑程度，取值越小，字体越圆滑
            0.4f,    //文本在 Z 轴方向上的厚度
            &m_pMesh, NULL, NULL );
    }
```

在 CD3DWnd::Cleanup 中释放新增加的 COM 接口：

（D3DWnd.cpp）

```
... ...
m_pMesh->Release();
m_p2DFont->Release();     //释放二维文本的字体接口
m_pDevice->Release();
... ...
```

由于用 D3DXCreateText 生成的文本 Mesh 模型，其大小是一个固定值，和所用字体的尺寸无关，因此只能通过世界座标变换来调整文本的大小。修改函数 CD3DWnd::SetupMatrices，把文本放大两倍：

（D3DWnd.cpp）

```
... ...
::D3DXMatrixRotationY( &matRotate, angle );
//计算缩放变换矩阵：放大 2 倍
::D3DXMatrixScaling( &matZoom, 2.0f, 2.0f, 2.0f );
::D3DXMatrixMultiply( &matWorld, &matZoom, &matRotate );
... ...
```

最后在函数 CD3DWnd::Render 中加入二维及三维文本的显示代码：

（D3DWnd.cpp）

```
... ...
m_pDevice->BeginScene();
SetupMatrices();
SetLight();
//在窗口左上角显示二维文本
m_p2DFont->DrawText(
```

```
        "二维文本",        //文本内容
        8,                 //文本长度
        CRect(0,0,100,50), //文本的显示区域，使用窗口坐标
        DT_LEFT,           //显示格式：左对齐
        D3DCOLOR_XRGB(255,0,0) ); //文本颜色：红色
    //设置三维文本所用的材质：黄色
    D3DMATERIAL9 mtrl;
    ::ZeroMemory( &mtrl, sizeof(D3DMATERIAL9) );
    mtrl.Diffuse.r = mtrl.Ambient.r = 1.0f;
    mtrl.Diffuse.g = mtrl.Ambient.g = 1.0f;
    mtrl.Diffuse.b = mtrl.Ambient.b = 0.0f;
    mtrl.Diffuse.a = mtrl.Ambient.a = 1.0f;
    m_pDevice->SetMaterial( &mtrl );
    //显示三维文本
    m_pMesh->DrawSubset(0);
    m_pDevice->EndScene();
    ... ...
```

编译运行程序，显示效果如图 31 所示。


图 31

**10 Direct3D 中的 2D**

前面介绍的都是三维图形的绘制，那么如何绘制二维图形呢？在常规的 Windows 编程中，系统提供了大量诸如画线、画圆的 Win32 GDI 函数，但是 Direct3D 没有这些类似的函数，二维图形也必须使用第 4.1.3 节介绍的那 6 种图元来构建。

Direct3D 提供了一种坐标格式 D3DFVF_XYZRHW，用于变换后的坐标空间。它有 4 个分量 x、y、z、rhw，其中 x、y 使用窗口坐标，z、rhw 代表深度。只要把 z、rhw 设置成常数（通常 z 取 0，rhw 取 1），就可以用它来定义二维图形的顶点。在使用这种坐标格式绘制二维图形时，建议用顶点颜色或纹理进行渲染，不要使用光照和材质，而且要禁用 Z 缓存，此时后绘制的图形将覆盖在最上面。

下面来演示如何在绘制三维图形的同时，显示一个二维的红色三角形和一个位图：前者使用顶点颜色渲染，图元格式选三角形列；后者用两个三角形拼成一个矩形，借助纹理实现位图的显示，图元格式选三角形带。

打开上一节的例程，在 D3DWnd.cpp 中定义二维图形的 FVF 顶点格式：

（D3DWnd.cpp）

```
... ...
#include "D3DWnd.h"
//红色三角形的 FVF 格式：RHW 坐标、顶点颜色
struct CUSTOMVERTEX3
{
    float x,y,z,rhw;        //顶点坐标
    DWORD color;            //顶点颜色
};
#define D3DFVF_CUSTOMVERTEX3 (D3DFVF_XYZRHW | D3DFVF_DIFFUSE)
//位图矩形的 FVF 格式：RHW 坐标、顶点颜色、纹理坐标
struct CUSTOMVERTEX4
{
    float x,y,z,rhw;        //顶点坐标
    DWORD color;            //顶点颜色
    float tu,tv;            //纹理坐标
};
#define D3DFVF_CUSTOMVERTEX4 (D3DFVF_XYZRHW | D3DFVF_DIFFUSE | D3DFVF_TEX1)
... ...
```

为 CD3DWnd 添加下列数据成员：

（D3DWnd.h）

```
... ...
LPD3DXFONT m_p2DFont; //用于显示二维文本的字体接口指针

LPDIRECT3DVERTEXBUFFER9 m_pVB3;//红色三角形的顶点缓存区接口指针
LPDIRECT3DVERTEXBUFFER9 m_pVB4;//位图矩形的顶点缓存区接口指针
LPDIRECT3DTEXTURE9 m_pTexture;      //位图矩形的纹理对象接口指针
... ...
```

在 CD3DWnd::InitGeometry 中创建二维图形的顶点缓存区和纹理，以下假设要显示的位

图文件 leaves.jpg 位于源程序所在目录：

（D3DWnd.cpp）

```
... ...
//创建三维文本的 Mesh 对象
::D3DXCreateText(... ...
//建立红色三角形的数学模型
CUSTOMVERTEX3 vertices3[3] =        //顶点为红色
    {{ 10, 30, 0.0f, 1.0f, D3DCOLOR_XRGB(255,0,0) },
    { 110, 30, 0.0f, 1.0f, D3DCOLOR_XRGB(255,0,0) },
    { 10, 130, 0.0f, 1.0f, D3DCOLOR_XRGB(255,0,0) }};
//创建红色三角形的顶点缓存区，填入顶点数据
m_pDevice->CreateVertexBuffer(
    sizeof(vertices3), 0, D3DFVF_CUSTOMVERTEX3,
    D3DPOOL_DEFAULT, &m_pVB3, NULL );
void* pVertices;
m_pVB3->Lock( 0, sizeof(vertices3), (void**)&pVertices, 0 );
memcpy( pVertices, vertices3, sizeof(vertices3) );
m_pVB3->Unlock();
//建立位图矩形的数学模型
CUSTOMVERTEX4 vertices4[4] =        //顶点为白色
    {{ 20, 80, 0.0f, 1.0f, D3DCOLOR_XRGB(255,255,255), 0.0f, 0.0f },
    { 120, 80, 0.0f, 1.0f, D3DCOLOR_XRGB(255,255,255), 1.0f, 0.0f },
    { 20, 180, 0.0f, 1.0f, D3DCOLOR_XRGB(255,255,255), 0.0f, 1.0f },
    { 120, 180, 0.0f, 1.0f, D3DCOLOR_XRGB(255,255,255), 1.0f, 1.0f }};
//创建位图矩形的顶点缓存区，填入顶点数据
m_pDevice->CreateVertexBuffer(
    sizeof(vertices4), 0, D3DFVF_CUSTOMVERTEX4,
    D3DPOOL_DEFAULT, &m_pVB4, NULL );
m_pVB4->Lock( 0, sizeof(vertices4), (void**)&pVertices, 0 );
memcpy( pVertices, vertices4, sizeof(vertices4) );
m_pVB4->Unlock();
//创建纹理对象
::D3DXCreateTextureFromFile( m_pDevice, "leaves.jpg", &m_pTexture );
... ...
```

在 CD3DWnd::Cleanup 中释放新增加的 COM 接口：

（D3DWnd.cpp）

```
... ...
m_pTexture->Release();      //释放纹理对象
m_pVB3->Release();          //释放红色三角形的顶点缓存区
m_pVB4->Release();          //释放位图矩形的顶点缓存区
m_pMesh->Release();
... ...
```

最后 CD3DWnd::Render 中添加二维图形的渲染代码：

（D3DWnd.cpp）

```
... ...
//显示三维文本
m_pMesh->DrawSubset(0);
//因为二维图形使用顶点颜色或纹理渲染，所以关闭光照
m_pDevice->SetRenderState( D3DRS_LIGHTING, FALSE );
//关闭 Z 缓存
m_pDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_FALSE );
//显示红色三角形
m_pDevice->SetFVF( D3DFVF_CUSTOMVERTEX3 );
m_pDevice->SetStreamSource( 0, m_pVB3, 0, sizeof(CUSTOMVERTEX3) );
m_pDevice->DrawPrimitive( D3DPT_TRIANGLELIST, 0, 1 );
//加载纹理，纹理混合方式使用缺省值。由于使用顶点颜色渲染，
//缺省的纹理混合方式＝纹理×顶点颜色。
m_pDevice->SetTexture( 0, m_pTexture );
//显示位图矩形
m_pDevice->SetFVF( D3DFVF_CUSTOMVERTEX4 );
m_pDevice->SetStreamSource( 0, m_pVB4, 0, sizeof(CUSTOMVERTEX4) );
m_pDevice->DrawPrimitive( D3DPT_TRIANGLESTRIP, 0, 2 );
m_pDevice->SetTexture( 0, NULL );
//二维图形绘制完毕，重新打开光照和 Z 缓存
m_pDevice->SetRenderState( D3DRS_LIGHTING, TRUE );
m_pDevice->SetRenderState( D3DRS_ZENABLE, D3DZB_TRUE );
m_pDevice->EndScene();
... ...
```

编译程序，然后从 VC6 的集成环境中运行，以确保当前目录为源程序目录，运行结果如图 32 所示。


图 32

**11 Direct3D 的程序结构**

在教程的最后，谈一下 Direct3D 的程序结构。

本文为了方便 VC 的初学者，使用 MFC 来建立 Direct3D 程序，但 DirectX SDK 提供的例程均使用 Win32 编程模式。在此模式下，用户需要自己编写（或管理）WinMain 主函数，并负责创建主窗口。下面是 WinMain 的程序结构：

```
... ...
//创建主窗口，也就是 Direct3D 窗口
HWND hWnd = CreateWindow(... ...
//初始化 Direct3D
InitD3D();
//建模
InitGeometry();
//进入消息循环
MSG msg;
ZeroMemory( &msg, sizeof(msg) );
while( msg.message!=WM_QUIT )
{
    if( PeekMessage( &msg, NULL, 0U, 0U, PM_REMOVE ) )
    {   //如果有消息，则处理消息
        TranslateMessage( &msg );
        DispatchMessage( &msg );
    }
    else
        //如果没有消息，就调用 Render
        Render();
}
//释放接口
Cleanup();
... ...
```

采用这种结构时，只要程序处于空闲（Idle）状态，就会调用渲染函数，运行效率比 MFC 好得多，适合编写对实时性要求比较高的程序。

此前的例程都是借助定时器产生的 WM_TIMER 消息来控制实体的运动，改用 Win32 模式后，Render 的调用是随机发生的，那么如何控制实体运动呢？VC 提供了一个函数 clock，用于返回程序的运行时间。我们在 Render 中调用 clock，把它的返回值做为实体的运动时间，然后通过运动学方程推算出实体位置。

**附注**

[1] 我已经把本文中的所有例程及.x 格式转换工具放在互联网上，下载地址：http://oldsong.nease.net/d3d9-example.zip

[2] 本文中的插图 3 至 6、8、10、14、16、18、19、20、26 均引自 DirectX9 SDK 文档。